ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Λίστες ταυτοχρόνως προσβάσιμες.

Συγκριτική μελέτη της απόδοσης και υλοποίησης ταυτόχρονων δομών δεδομένων χρησιμοποιώντας mutex και spinlock locks, lock-free και transactional memory σε C++.

Concurrently accessed lists.

A comparative study between performance and implementation of concurrent data structures, using mutex and spinlock locks, lock-free and transactional memory in C++.

Διπλωματική Εργασία

Κωνσταντίνος Χ. Παλαιοδήμος

Επιβλέποντες Καθηγητές : Σταμούλης Γεώργιος, Καθηγητής Π.Θ.
Ευμορφόπουλος Νέστωρ, Επίκουρος Καθηγητής Π.Θ.

Βόλος, Μάρτιος 2018

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

UNIVERSITY OF THESSALY

DEPARTMENT OF ELECTRICAL & COMPUTER ENGINEERING

Concurrently accessed lists.

A comparative study between performance and implementation of concurrent data structures, using mutex and spinlock locks, lock-free and transactional memory in C++.

Graduate Thesis for the degree of

Diploma of Science in Electrical & Computer Engineering

By Konstantinos X. Palaiodimos

Head Supervisor

George Stamoulis

Second Supervisor

Nestor Eumorfopoulos

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

This page is intentionally left blank.

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

To my family and friends

Στην οικογένεια μου και στους φίλους μου

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

This page is intentionally left blank.

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

Ευχαριστίες

Με την περάτωση της παρούσας διπλωματικής εργασίας ολοκληρώνεται ο προπτυχιακός κύκλος σπουδών μου. Θα ήθελα λοιπόν, να ευχαριστήσω θερμά τους επιβλέποντες μου κ. Γεώργιο Σταμούλη και κ. Νέστωρα Ευμορφόπουλο, για την εμπιστοσύνη που επέδειξαν στο πρόσωπό μου με την ανάθεση του συγκεκριμένου θέματος, την άριστη συνεργασία και την συνεχή καθοδήγηση, η οποία διευκόλυνε την εκπόνηση της διπλωματικής εργασίας μου.

Τέλος, οφείλω ένα μεγάλο ευχαριστώ στην οικογένειά μου και στους φίλους μου για την αμέριστη υποστήριξη και την ανεκτίμητη βοήθεια που μου παρείχαν κατά την διάρκεια των σπουδών μου.

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

Contents

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

Περίληψη

Ο κύριος στόχος της παρούσας εργασίας ήταν η διερεύνηση των θεμάτων του ταυτόχρονου προγραμματισμού. Καθώς η εποχή των πολυπύρηνων επεξεργαστών έχει ήδη ανθίσει και οι αγορές τους κυριαρχούν, ο ταυτόχρονος προγραμματισμός φαίνεται να είναι η μοναδική λύση τόσο για υψηλή απόδοση όσο και για ενεργειακή απόδοση την ίδια στιγμή. Μετά την ανάλυση του ιστορικού υποβάθρου που οδήγησε σε αυτήν την εφεύρεση, ακολουθεί κάποια ξεχωριστή διαφοροποίηση μεταξύ των όρων ταυτόχρονου και παράλληλου προγραμματισμού. Επιπλέον, το θεωρητικό μέρος περιγράφει συνοπτικά το αρχιτεκτονικό υπόβαθρο και τους διαφορετικούς τύπους και επίπεδα παραλληλισμού, καθώς και τα βασικά προβλήματα ταυτόχρονου προγραμματισμού που πρέπει να αποφευχθούν. Τέλος, μια θεωρητική προσέγγιση στις ταυτόχρονες δομές δεδομένων και τους τρόπους εφαρμογής τους, συμπεριλαμβανομένων των locking, lock-free και transactional memory.

Το επόμενο παράρτημα περιέχει τις λεπτομέρειες της ρεαλιστικής υλοποίησης των ταυτόχρονων λιστών στο περιβάλλον της C ++. Πιο συγκεκριμένα, διεξάγεται διεξοδική ανάλυση σχετικά με τους τρόπους κατασκευής μιας ταυτόχρονης λίστας χρησιμοποιώντας locks, lock-free, την εφαρμογή αποδοτικών spinlocks και τη χρήση τους αντί για mutex locks ή ακόμα και με την αφαίρεση της transactional memory.

Ο απώτερος στόχος αυτής της βιβλιογραφικής ανασκόπησης είναι η ταξινόμηση των τρόπων μετασχηματισμού των λιστών σε ταυτόχρονη πρόσβαση βάσει της σύγκρισης και της αντίφασης όσον αφορά την απόδοση, την καταλληλότητα και την ευκολία υλοποίησης του καθενός.

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

Abstract

This thesis's main target was the exploration of concurrency matters. As the era of multicores has already risen and the markets are dominated by them, concurrency seems as the only solution towards both high performance and energy efficiency at the same time. After analyzing the history background that led to this invention, some distinctive differentiation between the terms concurrency and parallelism follows. Furthermore, the theoretical part contains briefly the architectural background and different types and levels of parallelism, and the basic concurrency problems that must be avoided. Finally, a theoretical approach to concurrent data structures and ways of implementing them including locking, lock-free and transactional memory.

The next session contains the details of the pragmatic implementation of concurrent lists in the environment of C++. More specifically, thorough analysis is done on the ways of constructing a concurrent list using locks, lock-free, implementing efficient spinlocks and using them instead of mutex locks or even expressed by the abstraction of transactional memory.

Ulterior aim of all this literature review, is the classification of ways to transform lists into concurrently accessed based on the comparison and contrast in terms of performance, suitability and ease of implementation of each one.

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

Introduction

Evolution of the technology alongside with the market.

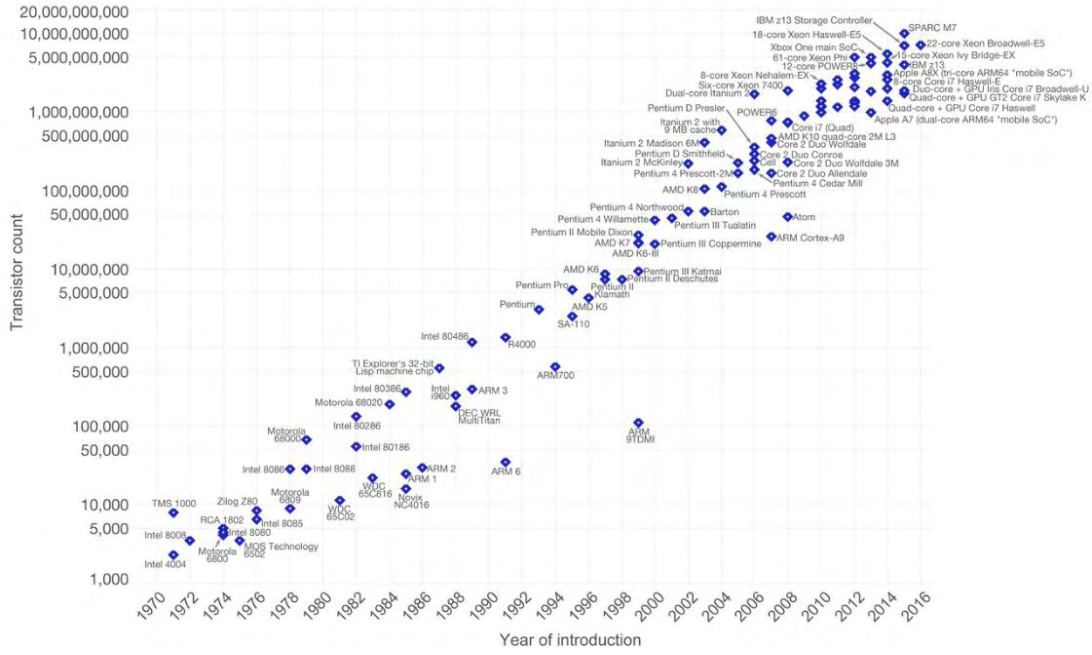
The microprocessor industry continues to have great importance in the course of technological advancements ever since their coming to existence in 1970s. The growing market and the demand for faster performance drove the industry to manufacture faster and smarter chips. One of the most classic and proven techniques to improve performance is to clock the chip at higher frequency which enables the processor to execute the programs in a much quicker time and the industry has been following this trend from 1983 – 2002. Additional techniques have also been devised to improve performance including parallel processing, data level parallelism and instruction level parallelism which have all proven to be very effective.[20] One such technique which improves significant performance boost is multi-core processors. Multi-core processors have been in existence since the past decade, but however have gained more importance off late due to technology limitations single-core processors are facing today such as high throughput and long-lasting battery life with high energy efficiency.

The struggle to keep up with Moore's law

Driven by a performance hungry market, microprocessors have always been designed keeping performance and cost in mind. Gordon Moore, founder of Intel Corporation predicted that the number of transistors on a chip will double once in every 18 months to meet this ever-growing demand which is popularly known as Moore's Law in the semiconductor industry. Advanced chip fabrication technology alongside with integrated circuit processing technology offers increasing integration density which has made it possible to integrate one billion transistors on a chip to improve performance. However, the performance increase by micro-architecture governed by Pollack's rule is roughly proportional to square root of increase in complexity. This would mean that doubling the logic on a processor core would only improve the performance by 40%. As advanced chip fabrication techniques come along another major bottleneck is discovered: power dissipation issue. Studies have shown that transistor leakage current increases as the chip size shrinks further and further which increases static power dissipation to large values.

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

One alternate means of improving performance is to increase the frequency of operation which enables faster execution of programs. However, the frequency is again limited to 4GHz currently as any increase beyond this frequency increases power dissipation again. “Battery life and system cost constraints drive the design team to consider power over performance in such a scenario”. Power consumption has increased to such high levels that traditional air-cooled microprocessor server boxes may require budgets for liquid-cooling or refrigeration hardware. Designers eventually hit what is referred to as the power wall, the limit on the amount of power a microprocessor could dissipate. Semiconductor industry once driven by performance being the major design objective, is today being driven by other important considerations such chip fabrication costs, fault tolerance, power efficiency and heat dissipation. This led to the development of multi-core processors which have been effective in addressing these challenges.



Moore's law describes the empirical regularity that the number of transistors on integrated circuits double approximately every two years. This advancement is important as other aspects of technological progress - such as processing speed or the price of electronic products - are strongly linked to Moore’s Law.

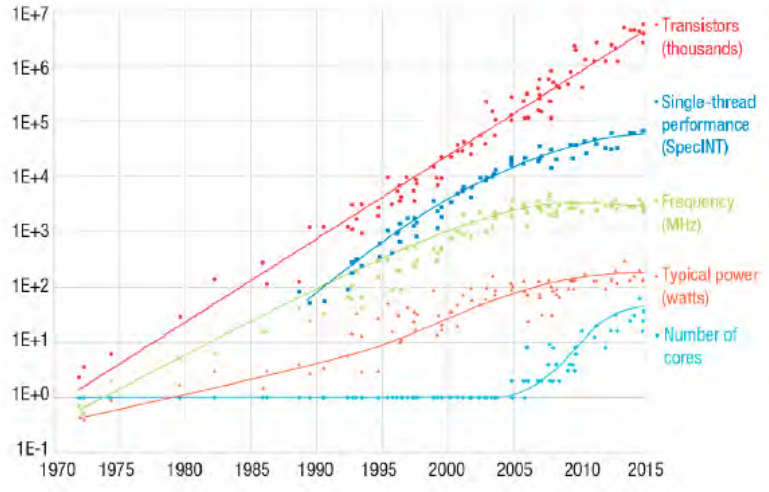
Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

The rise of the solution called multi-core

"A Multi-core processor is typically a single processor which contains several cores on a chip". The cores are functional units made up of computation units and caches. These multiple cores on a single chip combine to replicate the performance of a single faster processor. The individual cores on a multi-core processor don't necessarily run as fast as the highest performing single-core processors, but they improve overall performance by handling more tasks in parallel.

The performance boost can be seen by understanding the manner in which single core and multi-core processors execute programs. Single core processors running multiple programs would assign time slice to work on one program and then assign different time slices for the remaining programs. If one of the processes is taking longer time to complete then all the rest of the processes start lagging behind. However, In the case of multi-core processors if you have multiple tasks that can be run in parallel at the same time, each of them will be executed by a separate core in parallel thus boosting the performance. The multiple cores inside the chip are not clocked at a higher frequency, but instead their capability to execute programs in parallel is what ultimately contributes to the overall performance making them more energy efficient and low power cores as shown.

Multi-core processors are generally designed partitioned so that the unused cores can be powered down or powered up as and when needed by the application contributing to overall power dissipation savings.



Evolution of micro-processor performance over time. From 2000-2005 we see the end of "Dennard's scaling", leading to stagnation of the frequency of the cores and the rise of multi then many-core era.

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

Challenges in the multi-core era

Despite the many advantages that multi-core processors come with, there are a few major challenges the technology is facing. One main issue seen is with regard to software programs which run slower on multicore processors when compared to single core processors. It has been correctly pointed out that "Applications on multi-core systems don't get faster automatically as cores are increased". Programmers must write applications that exploit the increasing number of processors in a multi-core environment without stretching the time needed to develop software. Majority of applications used today were written to run on only a single processor, failing to use the capability of multi-core processors.

Secondly, on-chip interconnects are becoming a critical bottle-neck in meeting performance of multi-core chips. With increasing number of cores comes along the huge interconnect delays (wire delays) when data has to be moved across the multi-core chip from memories in particular. The performance of the processor truly depends on how fast a CPU can fetch data rather than how fast it can operate on it to avoid data starvation scenario. Buffering and smarter integration of memory and processors are a few classic techniques which have attempted to address this issue.

Furthermore, increased design complexity due to possible race conditions as the number of cores increase in a multi-core environment. "Multiple threads accessing shared data simultaneously may lead to a timing dependent error known as data race condition". [54] In a multi-core environment data structure is open to access to all other cores when one core is updating it. In the event of a secondary core accessing data even before the first core finishes updating the memory, the secondary core faults in some manner. Race conditions are especially difficult to debug and cannot be detected by inspecting the code, because they occur randomly. Special hardware requirement implementing mutually exclusion techniques have to be implemented for avoiding race conditions.

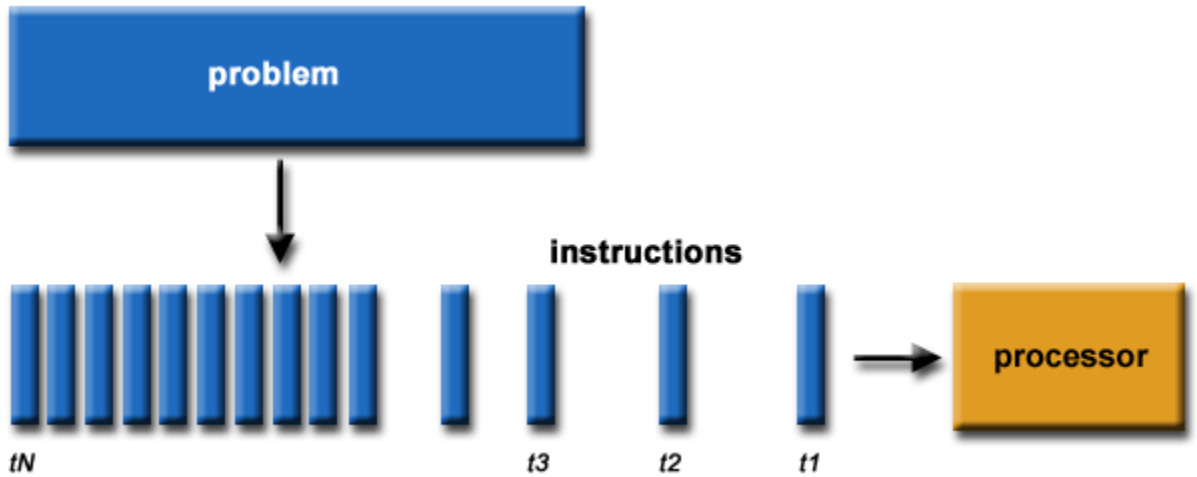
Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

Parallel Programming

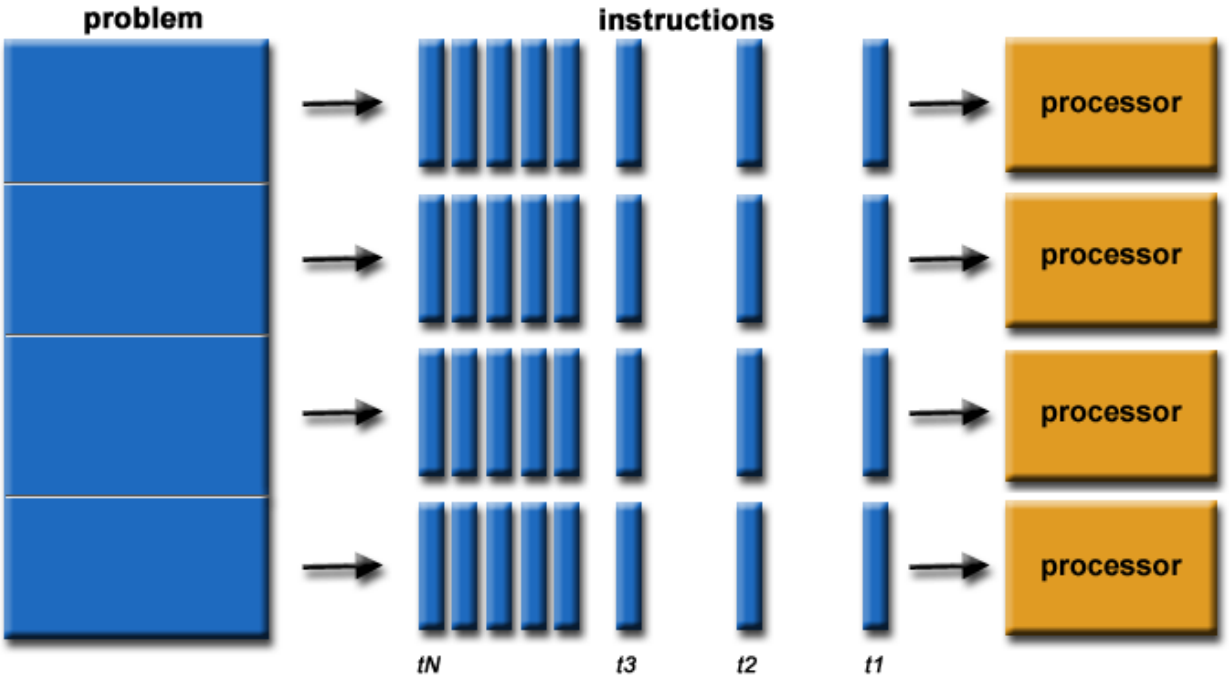
Traditionally, computer software has been written for serial computation. To solve a problem, an algorithm is constructed and implemented as a serial stream of instructions. These instructions are executed on a central processing unit on one computer. Only one instruction may execute at a time—after that instruction is finished, the next one is executed. [1]

Parallel computing, on the other hand, uses multiple processing elements simultaneously to solve a problem. This is accomplished by breaking the problem into independent parts so that each processing element can execute its part of the algorithm simultaneously with the others. The processing elements can be diverse and include resources such as a single computer with multiple processors, several networked computers, specialized hardware, or any combination of the above. [1]

Frequency scaling was the dominant reason for improvements in computer performance from the mid-1980s until 2004. The runtime of a program is equal to the number of instructions multiplied by the average time per instruction. Maintaining everything else constant, increasing the clock frequency decreases the average time it takes to execute an instruction. An increase in frequency thus decreases runtime for all compute-bound programs. [2]



Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7



Representation of the logic of parallel programming, where the problem is broken down into sub problems and assigned to many "workers".

Parallel computing is a type of computation in which many calculations or the execution of processes are carried out simultaneously.[3] Large problems can often be divided into smaller ones, which can then be solved at the same time. There are several different forms of parallel computing: bit-level, instruction-level, data, and task parallelism. Parallelism has long been employed in high-performance computing, but it's gaining broader interest due to the physical constraints preventing frequency scaling.[4] As power consumption (and consequently heat generation) by computers has become a concern in recent years,[5] parallel computing has become the dominant paradigm in computer architecture, mainly in the form of multi-core processors. [6]

Parallel computing is closely related to concurrent computing—they are frequently used together, and often conflated, though the two are distinct: it is possible to have parallelism without concurrency (such as bit-level parallelism), and concurrency without parallelism (such as multitasking by time-sharing on a single-core CPU).[7] In parallel computing, a computational task is typically broken down in several, often many, very similar subtasks that can be processed independently and whose results are combined afterwards, upon completion. In contrast, in concurrent computing, the various processes often do not address related tasks; when they do, as is typical in distributed computing, the separate tasks may have a

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

varied nature and often require some inter-process communication during execution.

Parallel computers can be roughly classified according to the level at which the hardware supports parallelism, with multi-core and multi-processor computers having multiple processing elements within a single machine, while clusters, MPPs, and grids use multiple computers to work on the same task. Specialized parallel computer architectures are sometimes used alongside traditional processors, for accelerating specific tasks.

In some cases parallelism is transparent to the programmer, such as in bit-level or instruction-level parallelism, but explicitly parallel algorithms, particularly those that use concurrency, are more difficult to write than sequential ones,[8] because concurrency introduces several new classes of potential software bugs, of which race conditions are the most common. Communication and synchronization between the different subtasks are typically some of the greatest obstacles to getting good parallel program performance.

A theoretical upper bound on the speed-up of a single program as a result of parallelization is given by Amdahl's law.

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

Concurrent Programming

Concurrent computing is a form of computing in which several computations are executed during overlapping time periods—*concurrently*—instead of *sequentially* (one completing before the next starts). This is a property of a system—this may be an individual program, a computer, or a network—and there is a execution point or "thread of control" for each computation ("process"). A *concurrent system* is one where a computation can advance without waiting for all other computations to complete. [9]

As a programming paradigm, concurrent computing is a form of modular programming, namely factoring an overall computation into sub computations that may be executed concurrently. Pioneers in the field of concurrent computing include Edsger Dijkstra, Per Brinch Hansen, and C.A.R. Hoare.

The concept of concurrent computing is frequently confused with the related separate but distinct concept of parallel computing,[7] although both can be described as "multiple processes executing *during the same period of time*". In parallel computing, execution occurs at the same physical instant: for example, on separate processors of a multi-processor machine, with the goal of speeding up computations—parallel computing is impossible on a (one-core) single processor, as only one computation can occur at any instant (during any single clock cycle). By contrast, concurrent computing consists of process *lifetimes* overlapping, but execution need not happen at the same instant. The goal here is to model processes in the outside world that happen concurrently, such as multiple clients accessing a server at the same time. Structuring software systems as composed of multiple concurrent, communicating parts can be useful for tackling complexity, regardless of whether the parts can be executed in parallel.[10]

For example, concurrent processes can be executed on one core by interleaving the execution steps of each process via time-sharing slices: only one process runs at a time, and if it does not complete during its time slice, it is *paused*, another process begins or resumes, and then later the original process is resumed. In this way, multiple processes are part-way through execution at a single instant, but only one process is being executed at that instant.

Concurrent computations may be executed in parallel, for example, by assigning each process to a separate processor or processor core, or distributing a computation across a network. In general, however, the languages, tools, and

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

techniques for parallel programming might not be suitable for concurrent programming, and vice versa.

The exact timing of when tasks in a concurrent system are executed depend on the scheduling, and tasks need not always be executed concurrently. For example, given two tasks, T1 and T2:

- T1 may be executed and finished before T2 or vice versa (serial and sequential)
- T1 and T2 may be executed alternately (serial and concurrent)
- T1 and T2 may be executed simultaneously at the same instant of time (parallel and concurrent)

Advantages of concurrent computing:

- Increased program throughput—parallel execution of a concurrent program allows the number of tasks completed in a given time to increase.
- High responsiveness for input/output—input/output-intensive programs mostly wait for input or output operations to complete. Concurrent programming allows the time that would be spent waiting to be used for another task.
- More appropriate program structure—some problems and problem domains are well-suited to representation as concurrent tasks or processes.

There are several reasons for a programmer to be interested in concurrency: [11]

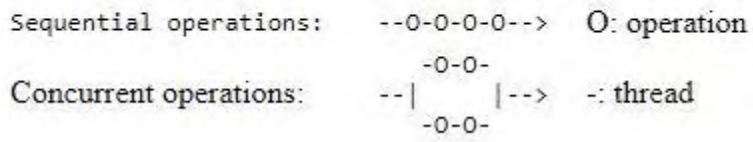
To better understand computer architecture (it has a great deal of concurrency with pipelining (multiple steps) and super-scalar (multiple instructions)) and

1. compiler design,
2. Some problems are most naturally solved by using a set of co-operating processes,
3. A sequential solution constitutes over specification, and
4. to reduce the execution time.

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

At the machine level, operations are *sequential*, if they occur one after the other, ordered in time. Operations are *concurrent*, if they overlap in time. In Figure 1, sequential operations are connected by a single thread of control while concurrent operations have multiple threads of control.

Figure 1: **Sequential and Concurrent Operations**

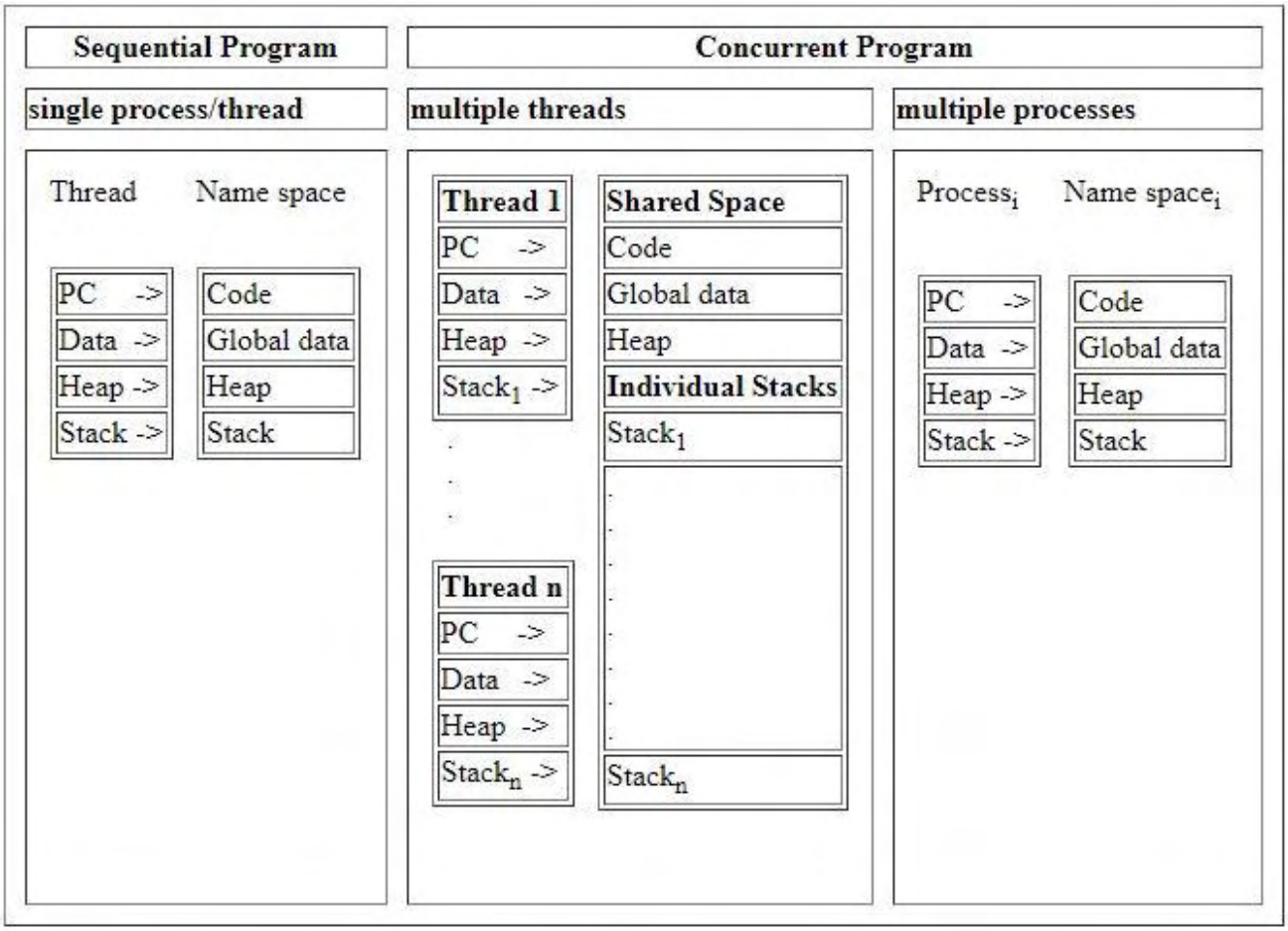


Operations in the source text of a program are *concurrent* if they could be, but need not be, executed in parallel. Thus concurrency occurs in a programming language when two or more operations could be but need not be executed in parallel. In Figure 2a the second assignment depends on the outcome of the first assignment while in Figure 2b neither assignment depends on the other and may be executed concurrently.

Figure 2: **Sequential and Concurrent Code**

a. not concurrent	b. concurrent
X := 5;	X := A*B + C;
Y := 3*X + 4	Y := 3*A + 7;

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7



Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

Speedup

In computer architecture, speedup is a process for increasing the performance between two systems processing the same problem. More technically, it is the improvement in speed of execution of a task executed on two similar architectures with different resources. The notion of speedup was established by Amdahl's law, which was particularly focused on parallel processing. However, speedup can be used more generally to show the effect on performance after any resource enhancement. [14]

Speedup can be defined for two different types of quantities: latency and throughput.

Latency of an architecture is the reciprocal of the execution speed of a task:

$$L = \frac{1}{v} = \frac{T}{W},$$

where

υ is the execution speed of the task;

T is the execution time of the task;

W is the execution workload of the task.

Throughput of an architecture is the execution rate of a task:

$$Q = \rho v A = \frac{\rho A W}{T} = \frac{\rho A}{L},$$

where

ρ is the execution density (e.g., The number of stages in an instruction pipeline for a pipelined architecture);

A is the execution capacity (e.g., the number of processors for a parallel architecture).

Latency is often measured in seconds per unit of execution workload. Throughput is often measured in units of execution workload per second. Another

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

unit of throughput is instructions per cycle (IPC) and its reciprocal, cycles per instruction (CPI) is another of unit of latency.

Speedup is dimensionless and defined differently for each type of quantity so that it is a consistent metric.

Speedup in latency is defined by the following formula:

$$S_{\text{latency}} = \frac{L_1}{L_2},$$

Speedup in throughput is defined by the following formula:

$$S_{\text{throughput}} = \frac{Q_2}{Q_1}$$

Simplified, given the old execution time Told and the new execution time Tnew for a program, the speedup is

Sp =T old/T new

Amdahl's Law

In computer architecture, Amdahl's law (or Amdahl's argument) is a formula which gives the theoretical speedup in latency of the execution of a task at fixed workload that can be expected of a system whose resources are improved. It is named after computer scientist Gene Amdahl, and was presented at the AFIPS Spring Joint Computer Conference in 1967.

Amdahl's law is often used in parallel computing to predict the theoretical speedup when using multiple processors. For example, if a program needs 20 hours using a single processor core, and a particular part of the program which takes one hour to execute cannot be parallelized, while the remaining 19 hours (p = 0.95) of execution time can be parallelized, then regardless of how many processors are devoted to a parallelized execution of this program, the minimum execution time cannot be less than that critical one hour. Hence, the theoretical speedup is limited to at most 20 times (1/ (1 − p) = 20). For this reason, parallel computing with many processors is useful only for highly parallelizable programs. [14]

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

For over a decade prophets have voiced the contention that the organization of a single computer has reached its limits and that truly significant advances can be made only by interconnection of a multiplicity of computers in such a manner as to permit co-operative solution...The nature of this overhead (in parallelism) appears to be sequential so that it is unlikely to be amenable to parallel processing techniques. Overhead alone would then place an upper limit on throughput of five to seven times the sequential processing rate, even if the housekeeping were done in a separate processor...At any point in time it is difficult to foresee how the previous bottlenecks in a sequential computer will be effectively overcome.

Gene Amdahl 1967

Given

B E [0, 1], the fraction of an algorithm that is strictly serial,

n E N, the number of threads of execution,

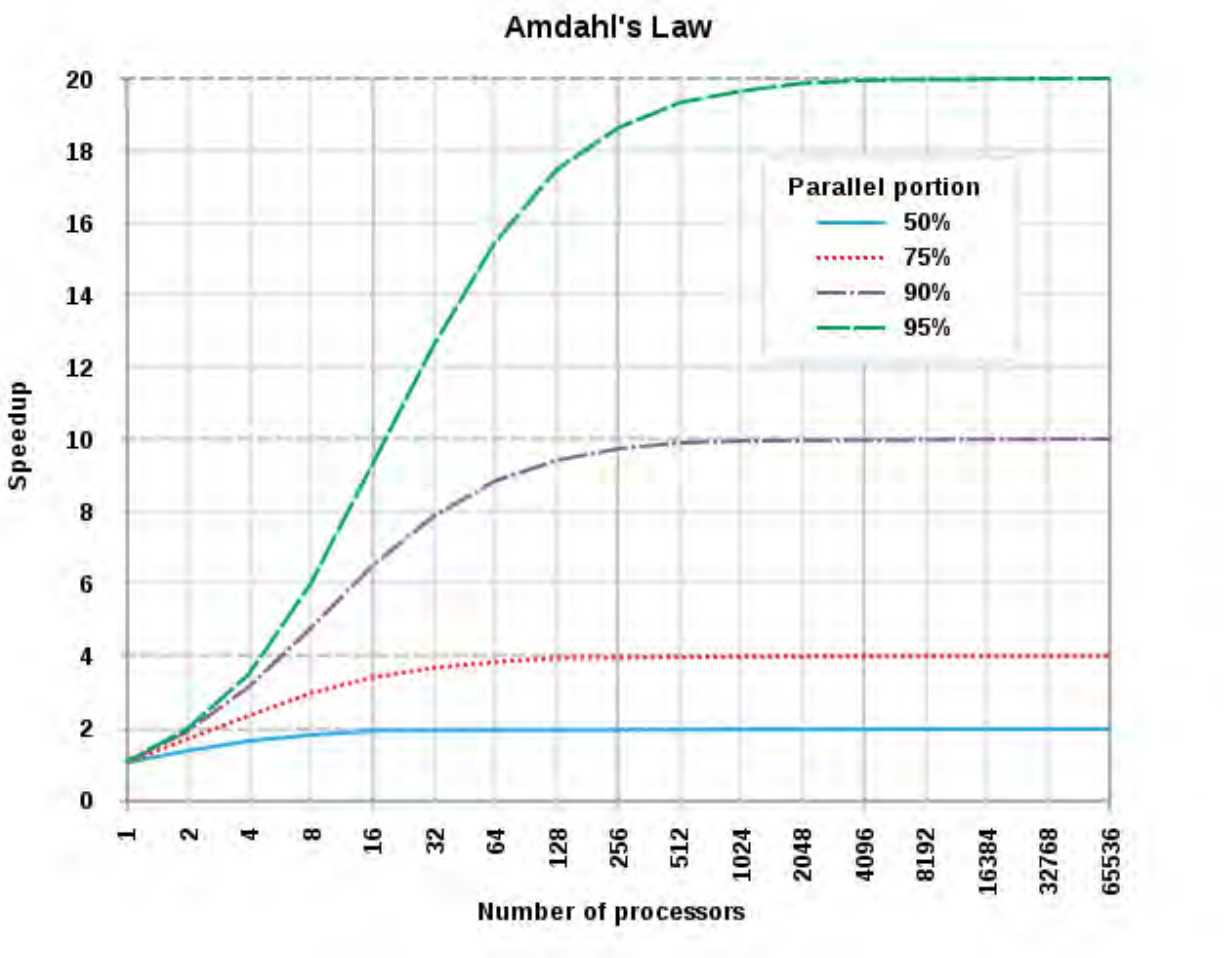
the time that it takes the algorithm to finish when executed on n threads is

$$T(n) = T(1)\left(B + \frac{1}{n}(1 - B)\right)$$

Consequently, the corresponding speedup is

$$S(n) = \frac{T(1)}{T(n)} = \frac{1}{B + \frac{1}{n}(1 - B)}$$

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7



Parallel Slowdown

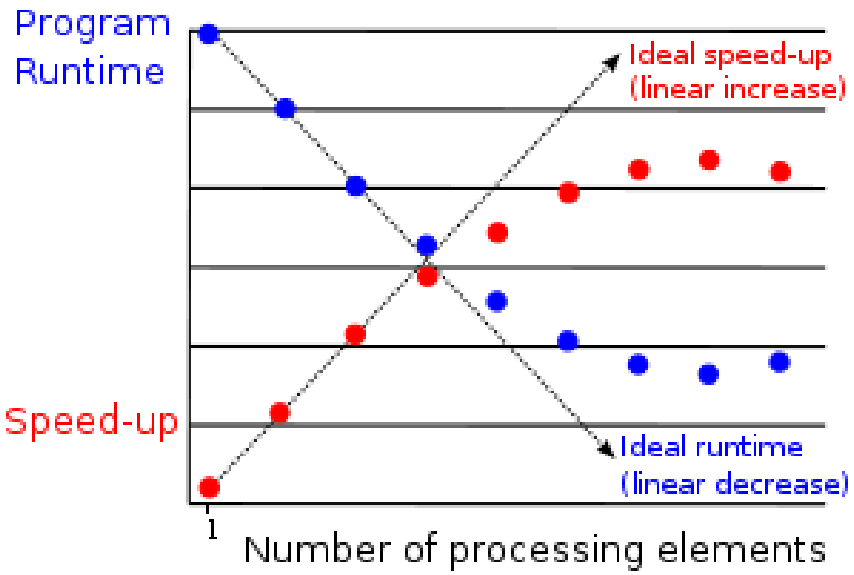
Parallel slowdown is a phenomenon in parallel computing where parallelization of a parallel algorithm beyond a certain point causes the program to run slower (take more time to run to completion) [15]

Parallel slowdown is typically the result of a communications bottleneck. As more processor nodes are added, each processing node spends progressively more time doing communication than useful processing. At some point, the communications overhead created by adding another processing node surpasses the increased processing power that node provides, and parallel slowdown occurs.

Parallel slowdown occurs when the algorithm requires significant communication, particularly of intermediate results. Some problems, known as

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

embarrassingly parallel problems, do not require such communication, and thus are not affected by slowdown. [14]



Super-Linear Speedup

Superlinear speedup comes from exceeding naively calculated speedup even after taking into account the communication process (which is fading, but still this is the bottleneck).

For example we have a serial algorithm that takes 1t to execute. We have 1024 cores, so naive speedup is 1024x, or it takes t/1024, but it should be calculated from Amdahl's equation taking into account memory transfer, slight modifications to algorithm, parallelization time.

So speedup should be lower than 1024x, but sometimes it happens that speedup is bigger, then we call it superlinear.

This comes from vast amount of places: cache usage (what fit into registers, main memory or mass storage, where very often more processing units gives overall more registers per subtask), memory hit patterns, simply better (or a slight different) algorithm, flaws in the serial code.

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

For example random process that searches space for result is now divided into 1024 searchers covering more space at once so finding solution faster is more probable. There are byproducts (if we swap elements like in bubble sort and switch into GPU it swaps all pairs at once, while serial only up to point).

On the distributed system communication is even more costly, so programs are changed to make memory usage local (which also changes memory access, divides problem differently than in sequential application). And the most important, the sequential program is not ideally the same as parallel version - different technology, environment, algorithm etc. so it is hard to compare them. [16]

Theoretically speedup can never exceed the number of processing elements pp. If the best sequential algorithm takes Ts units of time to solve a given problem on a single processing element, then a speedup of p can be obtained on p processing elements if none of them spends more than time Ts/p. A speedup greater than p is possible only if each processing element spends less than time Ts/p solving the problem. In this case, a single processing element could emulate the p processing elements and solve the problem in fewer than Ts units of time. This is a contradiction because speedup, by definition is computed with respect to the best sequential algorithm. [17]

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

Architectures

The last decades of the 20th century it was urgent that new architectures of hardware would be designed so that they could handle parallel computing. The oldest and most popular attempt to classify the parallel architectures was Flynn's taxonomy. More recently, and trying to redress the inadequacy of Flynn's scheme, Handler's (Erlangen) classification came up.

Flynn's taxonomy

Flynn's taxonomy is a classification of computer architectures, proposed by Michael J. Flynn in 1966. The classification system has stuck, and has been used as a tool in design of modern processors and their functionalities. Since the rise of multiprocessing central processing units (CPUs), a multiprogramming context has evolved as an extension of the classification system. [14]

Any system is based upon two important elements:

1. Instructions and
2. Data.

The data elements are manipulated according to the instructions. Depending upon the number of instructions executed and data elements manipulated simultaneously, Flynn makes the following classification. [18]

Single instruction stream single data stream (SISD)

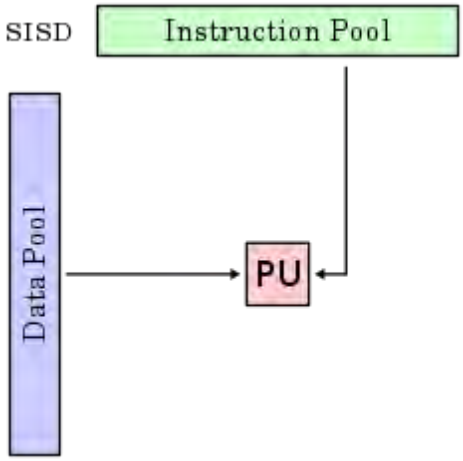
A sequential computer which exploits no parallelism in either the instruction or data streams. Single control unit (CU) fetches single instruction stream (IS) from memory. The CU then generates appropriate control signals to direct single processing element (PE) to operate on single data stream (DS) i.e., one operation at a time.

Examples of SISD architecture are the traditional uniprocessor machines like older personal computers (PCs; by 2010, many PCs had multiple cores) and mainframe computers. [14]

A single-instruction single-data machine is also commonly called a classical von Neumann Machine. These systems are separated into two divisions which are

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

the memory and the CPU (central processing unit). The memory portion holds both the program instructions and the data while the CPU interprets and executes the commands in the program. In the SISD model, the CPU is further divided into two more sections called the control unit and the arithmetic-logic unit (ALU). The control unit is in charge of executing the programs and the ALU does the actual computations called for by the program. Instructions on SISD machines are done in a sequential manner. [19]



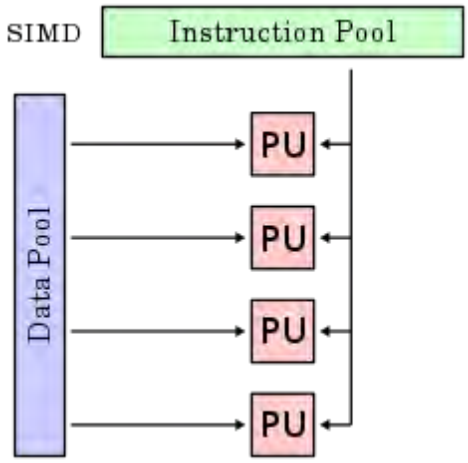
Single instruction stream, multiple data streams (SIMD)

A computer which exploits multiple data streams against a single stream to perform operations which may be naturally parallelized. For example, an array processor or graphics processing unit (GPU). [14] This system has only one CPU acting as the control unit and a number of ALUs which execute the given commands, with a limited amount of personal memory. The CPU will broadcast the same command to all the ALUs, which will either respond by computing or remain idle. [19] In the SIMD model there are two types of architectures:

1. Shared-memory model; and
2. Direct-connection networks.

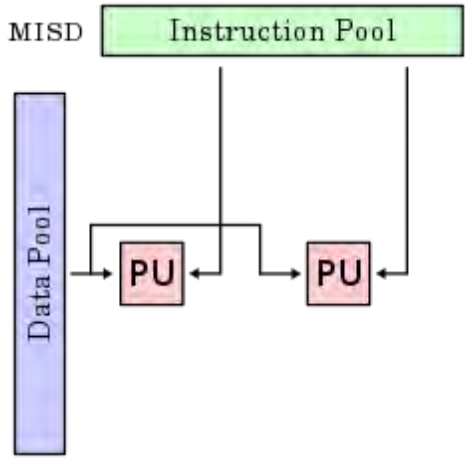
In the shared-memory model there is a common memory, which is share by all processors. Communication between the two processors takes place only through the shared memory. In the direct-connection network, independent processors are connected using wires, to any desired topologies such as rings, hyper cubes, and so on. [18]

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7



Multiple instruction streams, single data stream (MISD)

Multiple instructions operate on one data stream. This is an uncommon architecture which is generally used for fault tolerance. No computers have been designed so far to fit in this model. Heterogeneous systems operate on the same data stream and must agree on the result. Examples include the Space Shuttle flight control computer. [14] An example of an MISD architecture would be a system where each machine would perform different operations on the same data set. [19]

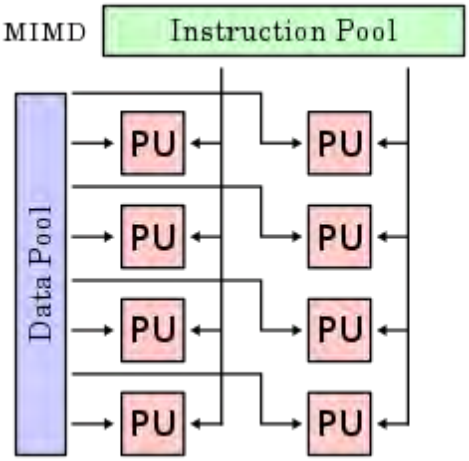


Multiple instruction streams, multiple data streams (MIMD)

Multiple autonomous processors (each have a control unit and an ALU) simultaneously executing different instructions on different data. MIMD

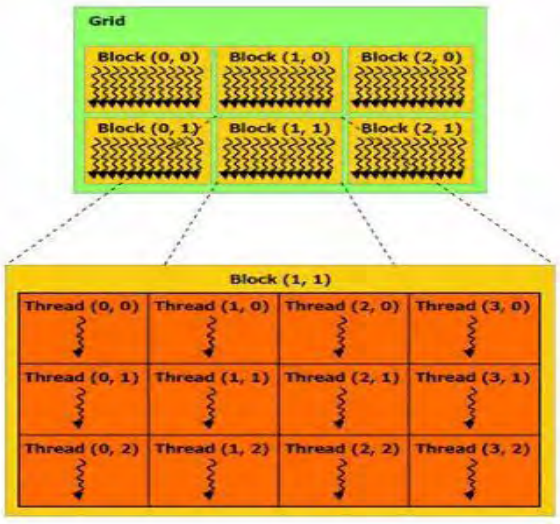
Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

architectures include multi-core superscalar processors, and distributed systems, using either one shared memory space or a distributed memory space. [14] This kind of system is considered asynchronous and only operates synchronously if specifically programed to operate that way. Since this design has separate instruction and data stream, it is well suited for a wide variety of applications. [19]



Single instruction, multiple threads (SIMT)

Single instruction, multiple threads (SIMT) is an execution model used in parallel computing where single instruction, multiple data (SIMD) is combined with multithreading. This is not originally part of Flynn's taxonomy but a proposed addition. [14]



Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

Primitive levels of Parallelism

Advances in technology determine what is possible; architecture translates the potential of the technology into performance and capability. There are fundamentally two ways in which a larger volume of resources, more transistors, improves performance: parallelism and locality. Moreover, these two fundamentally compete for the same resources. Whenever multiple operations are performed in parallel the number of cycles required to execute the program is reduced.

Examining the trends in microprocessor architecture will help build intuition towards the issues we will be dealing with in parallel machines. It will also illustrate how fundamental parallelism is to conventional computer architecture and how current architectural trends are leading toward multiprocessor designs. [21]

Bit-level Parallelism

The history of computer architecture has traditionally been divided into four generations identified by the basic logic technology: tubes, transistors, integrated circuits, and VLSI. Into the fourth or VLSI generation there has been tremendous architectural advance. The strongest delineation is the kind of parallelism that is exploited. The period up to about 1985 is dominated by advancements in bit-level parallelism, with 4-bit microprocessors replaced by 8-bit, 16-bit, and so on. Doubling the width of the datapath reduces the number of cycles required to perform a full 32-bit operation. This trend slows once a 32-bit word size is reached in the mid-80s, with only partial adoption of 64-bit operation obtained a decade later. Further increases in word-width will be driven by demands for improved floating-point representation and a larger address space, rather than performance. With address space requirements growing by less than one bit per year, the demand for 128-bit operation appears to be well in the future. The early microprocessor period was able to reap the benefits of the easiest form of parallelism: bit-level parallelism in every operation.

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

Instruction-Level Parallelism

The period from the mid-80s to mid-90s is dominated by advancements in instruction-level parallelism. Full word operation meant that the basic steps in instruction processing (instruction decode, integer arithmetic, and address calculation) could be performed in a single cycle; with caches the instruction fetch and data access could also be performed in a single cycle, most of the time. The RISC approach demonstrated that, with care in the instruction set design, it was straightforward to pipeline the stages of instruction processing so that an instruction is executed almost every cycle, on average. Thus the parallelism inherent in the steps of instruction processing could be exploited across a small number of instructions. While pipelined instruction processing was not new, it had never before been so well suited to the underlying technology. In addition, advances in compiler technology made instruction pipelines more effective.

However, increasing the amount of instruction level parallelism that the processor can exploit is only worthwhile if the processor can be supplied with instructions and data fast enough to keep it busy. In order to satisfy the increasing instruction and data bandwidth requirement, larger and larger caches were placed on-chip with the processor, further consuming the ever increasing number of transistors. With the processor and cache on the same chip, the path between the two could be made very wide to satisfy the bandwidth requirement of multiple instruction and data accesses per cycle. However, as more instructions are issued each cycle, the performance impact of each control transfer and each cache miss becomes more significant.

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

ILP: Implementation Techniques

How much ILP exists in programs is very application specific. In certain fields, such as graphics and scientific computing the amount can be very large. However, workloads such as cryptography may exhibit much less parallelism.

Micro-architectural techniques that are used to exploit ILP include:

- Instruction pipelining where the execution of multiple instructions can be partially overlapped.
- Superscalar execution, VLIW, and the closely related explicitly parallel instruction computing concepts, in which multiple execution units are used to execute multiple instructions in parallel.
- Out-of-order execution where instructions execute in any order that does not violate data dependencies.
- Register renaming which refers to a technique used to avoid unnecessary serialization of program operations imposed by the reuse of registers by those operations, used to enable out-of-order execution.
- Speculative execution which allow the execution of complete instructions or parts of instructions before being certain whether this execution should take place. A commonly used form of speculative execution is control flow speculation.
- Branch prediction which is used to avoid stalling for control dependencies to be resolved. Branch prediction is used with speculative execution.
- Dataflow architectures

It is known that the ILP is exploited by both the compiler and hardware support but the compiler also provides inherit and implicit ILP in programs to hardware by compilation optimization. Some optimization techniques for extracting available ILP in programs would include scheduling, register allocation/renaming, and memory access optimization. [14]

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

Thread-Level Parallelism

Task parallelism (also known as function parallelism and control parallelism) is a form of parallelization of computer code across multiple processors in parallel computing environments. Task parallelism focuses on distributing tasks—concurrently performed by processes or threads—across different processors. In contrast to data parallelism which involves running the same task on different components of data, task parallelism is distinguished by running many different tasks at the same time on the same data. [50] A common type of task parallelism is pipelining which consists of moving a single set of data through a series of separate tasks where each task can execute independently of the others.

In a multiprocessor system, task parallelism is achieved when each processor executes a different thread (or process) on the same or different data. The threads may execute the same or different code. In the general case, different execution threads communicate with one another as they work, but is not a requirement. Communication usually takes place by passing data from one thread to the next as part of a workflow. [51]

As a simple example, if a system is running code on a 2-processor system (CPUs "a" & "b") in a parallel environment and we wish to do tasks "A" and "B", it is possible to tell CPU "a" to do task "A" and CPU "b" to do task "B" simultaneously, thereby reducing the run time of the execution. The tasks can be assigned using conditional statements as described below.

Task parallelism emphasizes the distributed (parallelized) nature of the processing (i.e. threads), as opposed to the data (data parallelism). Most real programs fall somewhere on a continuum between task parallelism and data parallelism. [52]

Thread-level parallelism (TLP) is the parallelism inherent in an application that runs multiple threads at once. This type of parallelism is found largely in applications written for commercial servers such as databases. By running many threads at once, these applications are able to tolerate the high amounts of I/O and memory system latency their workloads can incur - while one thread is delayed waiting for a memory or disk access, other threads can do useful work.

The exploitation of thread-level parallelism has also begun to make inroads into the desktop market with the advent of multi-core microprocessors. This has occurred because, for various reasons, it has become increasingly impractical to

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

increase either the clock speed or instructions per clock of a single core. If this trend continues, new applications will have to be designed to utilize multiple threads in order to benefit from the increase in potential computing power. This contrasts with previous microprocessor innovations in which existing code was automatically sped up by running it on a newer/faster computer.

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

Memory Models

Basic differences in types of memory.

Data and code in a parallel program are stored in the main memory accessible for processors of the executive system. Regarding the way in which the main memory is used by processors in a multiprocessor system, we divide parallel systems onto shared memory system and distributed memory systems.

Shared memory

Shared memory is memory that may be simultaneously accessed by multiple programs with an intent to provide communication among them or avoid redundant copies. Using memory for communication inside a single program, e.g. among its multiple threads, is also referred to as shared memory.

Shared memory is an efficient means of passing data between programs. Depending on context, programs may run on a single processor or on multiple separate processors.

In a shared memory system, all processors can access all the main memory address space. Fragments of the address space are usually located in separate memory modules, which are supplied with separate address decoders. Communication between processors (program code fragments) is done by means of shared variables access in the main memory. It is called communication through shared variables. Fetching instructions for execution in processors is also done from a shared memory. The efficiency of accessing memory modules depends on the structure and properties of the interconnection network. This network is a factor, which imitates the memory access throughput for a larger number of processors. It sets a limit on the number of processors in such systems, with which good efficiency of a parallel system is achieved. Multiprocessor systems with shared memory are called tightly coupled systems or multiprocessors. Due to symmetric access of all processors to all memory modules, the computations in such systems are called Symmetric Multiprocessing - SMP.

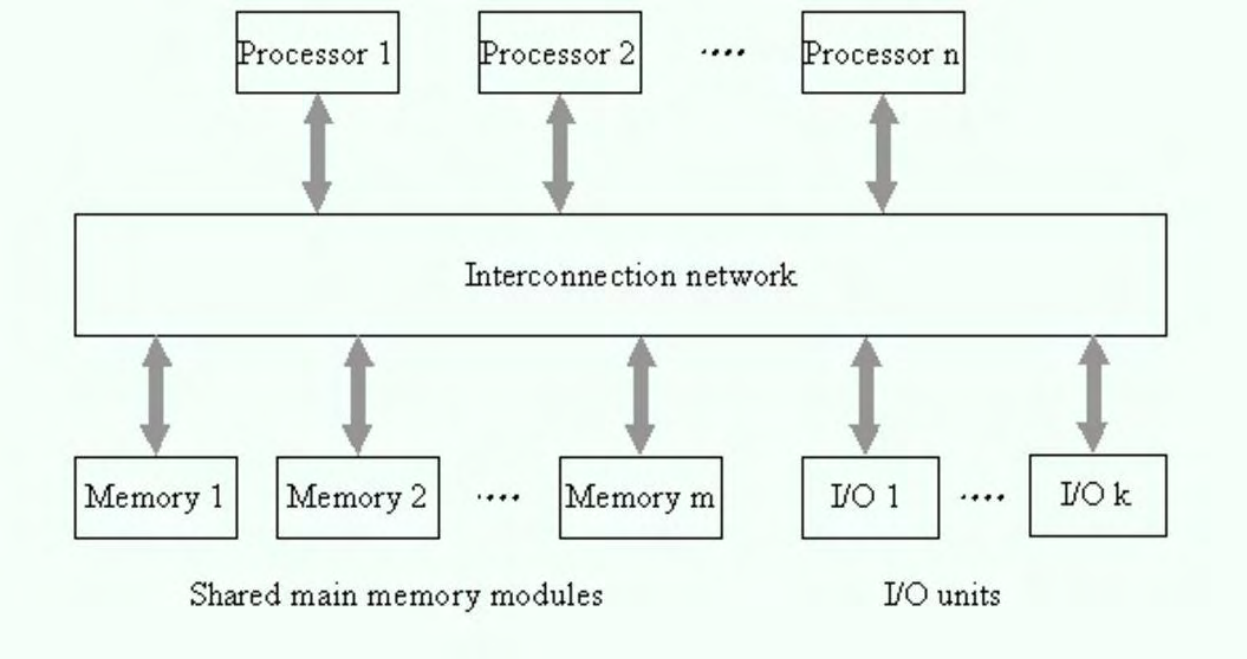
A shared memory system is relatively easy to program since all processors share a single view of data and the communication between processors can be as

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

fast as memory accesses to a same location. The issue with shared memory systems is that many CPUs need fast access to memory and will likely cache memory, which has two complications:

Access time degradation: when several processors try to access the same memory location it causes contention. Trying to access nearby memory locations may cause false sharing. Shared memory computers cannot scale very well. Most of them have ten or fewer processors;

Lack of data coherence: whenever one cache is updated with information that may be used by other processors, the change needs to be reflected to the other processors, otherwise the different processors will be working with incoherent data. Such cache coherence protocols can, when they work well, provide extremely high-performance access to shared information between multiple processors. On the other hand, they can sometimes become overloaded and become a bottleneck to performance.



A multiprocessor system with shared memory (tightly coupled system)

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

Distributed memory

Distributed memory refers to a multiprocessor computer system in which each processor has its own private memory. Computational tasks can only operate on local data, and if remote data is required, the computational task must communicate with one or more remote processors. In contrast, a shared memory multiprocessor offers a single memory space used by all processors. Processors do not have to be aware where data resides, except that there may be performance penalties, and that race conditions are to be avoided.

In a distributed memory system there is typically a processor, a memory, and some form of interconnection that allows programs on each processor to interact with each other. The interconnect can be organized with point to point links or separate hardware can provide a switching network.

In a distributed memory multiprocessor system, each processor has its local memory with the address space available only for this processor. Processors can exchange data through the interconnection network by means of communication through the message passing.

The instructions "send message" and "receive message" are used in programs for this purpose. The communication instructions send or receive messages with the use of identifiers of special elements (variables) are called communication channels.

The channels represent the use of connections that exist permanently (or are created in the interconnection network) between processors. There exist processors that are specially adapted for sending and receiving messages by the existence of communication links. Communication links can be serial or parallel. The number of communication links in such processors is from 4 to 6 (ex. transputer - 4 serial links, SHARC - a DSP (Data Signal Processor) from Analog Devices - 6 parallel links). Each link is supervised by an independent processor controller that organizes external data transmissions over the link. When a message is sent, it is fetched from the processor main memory. A message received from a link is next sent to the main memory. Multiprocessor systems that have distributed memory are called in the literature loosely coupled systems. In such systems, it is possible to organize many inter-processor connections at the same time. It provides high communication efficiency and, as a consequence, high efficiency of parallel

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

computations in processors (due to distribution of memory accesses), which gives rise to calling computations in such systems the Massively Parallel Processing - MPP.

Communication by message passing in such systems can be executed according to the synchronous or asynchronous communication model.

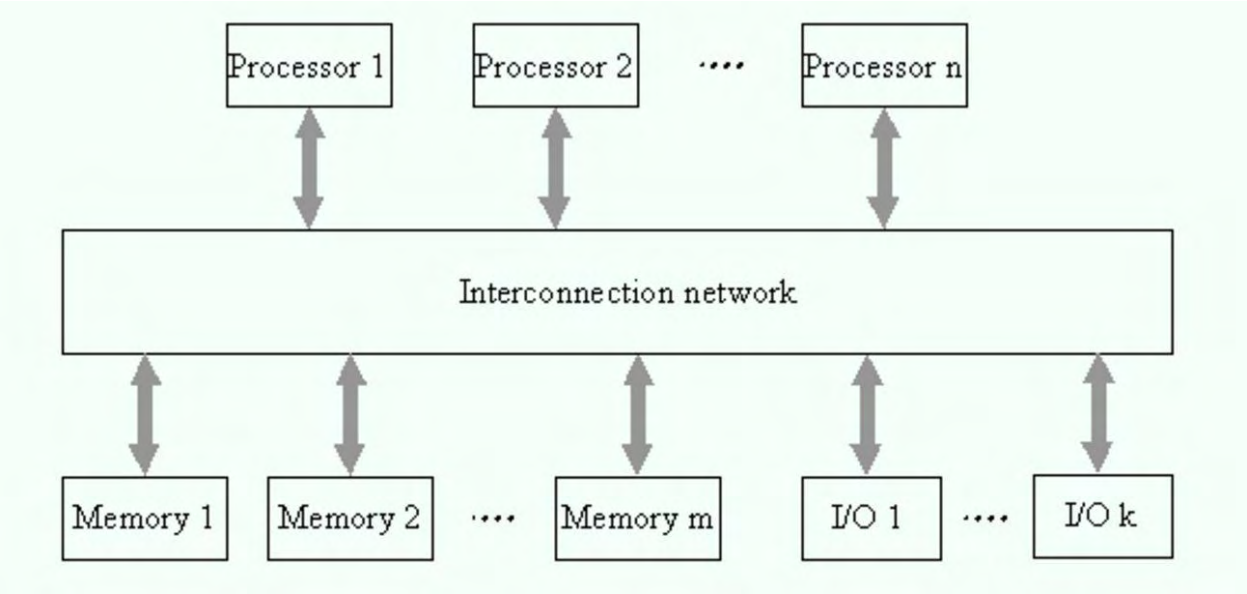
In the synchronous communication model, the partner processes (programs) - the sending and the receiving one, get synchronized on communication instructions in a given channel. It means that the sending process can start transmitting data only if the receiving process in another processor has reached execution of the receive instruction in the same channel as the sending one. Since the communication is performed with the use of send and receive instructions in both processors simultaneously, there is no need of buffering of messages, and so, they are sent as if they were sent directly from the main memory of one processor to the memory of the other one. All this is done under supervision of link controllers in both processors.

With the asynchronous communication model, the sending and receiving processes (programs) do not synchronize communication execution in the involved channels. A message is sent to a channel at any time and it is directed to the buffer for messages in a given channel in the controller at the other side of the interconnection between the processors. The receiving process reads the message from the buffer of the given channel at any convenient time.

The third type of multiprocessor systems are systems with the distributed shared memory called also the virtual shared memory. In such systems, which currently show strong development, each processor has a local main memory. However, each memory is placed in a common address space of the entire system. It means that each processor can have access to the local memory of any other processor. In this type of the system, communication between processors is done by accessing shared variables. It involves execution of a simple read or write instruction convening the shared variables in the memory of another processor. In each processor, a memory interface unit examines addresses used in current processor memory access instructions. As a result, it directs instruction execution to the local main memory bus or it sends the address together with the operation

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

code to the local memory interface of another processor. Sending the address and later the data is performed through the network that connects all processors



A multiprocessor system with a distributed memory (loosely coupled system)

Memory Access

Uniform memory access (UMA)

Uniform memory access (UMA) is a shared memory architecture used in parallel computers. All the processors in the UMA model share the physical memory uniformly. In a UMA architecture, access time to a memory location is independent of which processor makes the request or which memory chip contains the transferred data. In the UMA architecture, each processor may use a private cache. Peripherals are also shared in some fashion. The UMA model is suitable for general purpose and time sharing applications by multiple users. It can be used to speed up the execution of a single large program in time-critical applications.

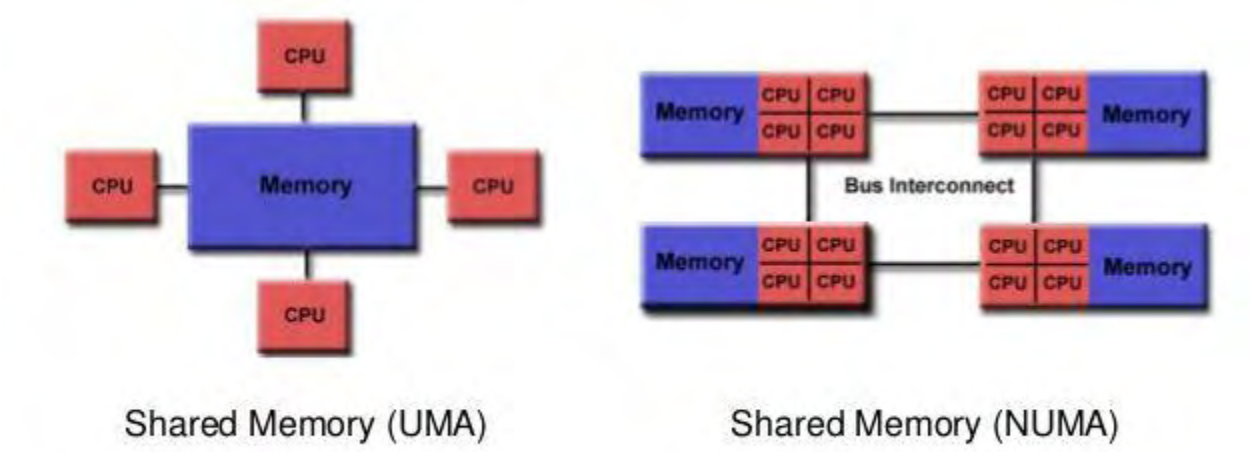
There are three types of UMA architectures:

- UMA using bus-based symmetric multiprocessing (SMP) architectures;

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

- UMA using crossbar switches;
- UMA using multistage interconnection networks. [25]

In April 2013, the term hUMA (*heterogeneous uniform memory access*) began to appear in AMD promotional material to refer to CPU and GPU sharing the same system memory via cache coherent views. Advantages include an easier programming model and less copying of data between separate memory pools. [26]



Non-uniform memory access (NUMA)

Non-uniform memory access (NUMA) is a computer memory design used in multiprocessing, where the memory access time depends on the memory location relative to the processor. Under NUMA, a processor can access its own local memory faster than non-local memory (memory local to another processor or memory shared between processors). The benefits of NUMA are limited to particular workloads, notably on servers where the data is often associated strongly with certain tasks or users.

NUMA architectures logically follow in scaling from symmetric multiprocessing (SMP) architectures. They were developed commercially during the 1990s by Burroughs (later Unisys), Convex Computer (later Hewlett-Packard), Honeywell Information Systems Italy (HISI) (later Groupe Bull), Silicon Graphics (later Silicon Graphics International), Sequent Computer Systems (later IBM), Data General (later EMC), and Digital (later Compaq, then HP,

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

now HPE). Techniques developed by these companies later featured in a variety of Unix-like operating systems, and to an extent in Windows NT. [53]

Modern CPUs operate considerably faster than the main memory they use. In the early days of computing and data processing, the CPU generally ran slower than its own memory. The performance lines of processors and memory crossed in the 1960s with the advent of the first supercomputers. Since then, CPUs increasingly have found themselves "starved for data" and having to stall while waiting for data to arrive from memory. Many supercomputer designs of the 1980s and 1990s focused on providing high-speed memory access as opposed to faster processors, allowing the computers to work on large data sets at speeds other systems could not approach.

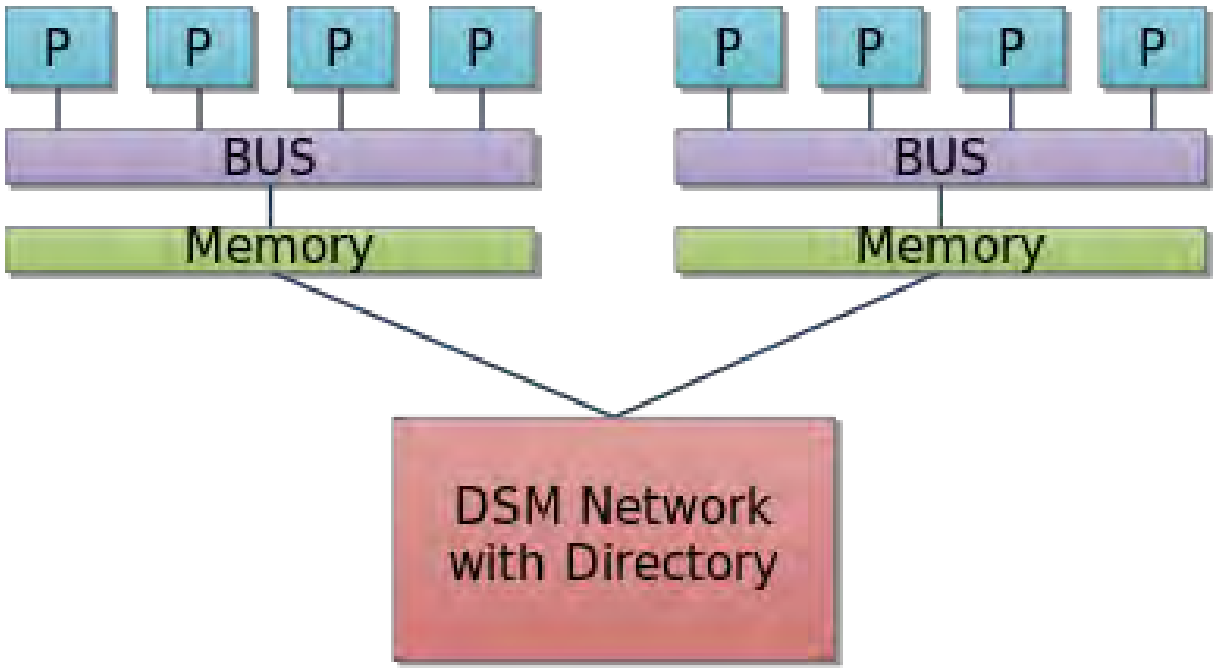
Limiting the number of memory accesses provided the key to extracting high performance from a modern computer. For commodity processors, this meant installing an ever-increasing amount of high-speed cache memory and using increasingly sophisticated algorithms to avoid cache misses. But the dramatic increase in size of the operating systems and of the applications run on them has generally overwhelmed these cache-processing improvements. Multi-processor systems without NUMA make the problem considerably worse. Now a system can starve several processors at the same time, notably because only one processor can access the computer's memory at a time. [27]

NUMA attempts to address this problem by providing separate memory for each processor, avoiding the performance hit when several processors attempt to address the same memory. For problems involving spread data (common for servers and similar applications), NUMA can improve the performance over a single shared memory by a factor of roughly the number of processors (or separate memory banks). [28]

Another approach to addressing this problem, used mainly in non-NUMA systems, is the multi-channel memory architecture, in which a linear increase in the number of memory channels increases the memory access concurrency linearly. [29]

Of course, not all data ends up confined to a single task, which means that more than one processor may require the same data. To handle these cases, NUMA systems include additional hardware or software to move data between memory banks. This operation slows the processors attached to those banks, so the overall speed increase due to NUMA depends heavily on the nature of the running tasks. [28]

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

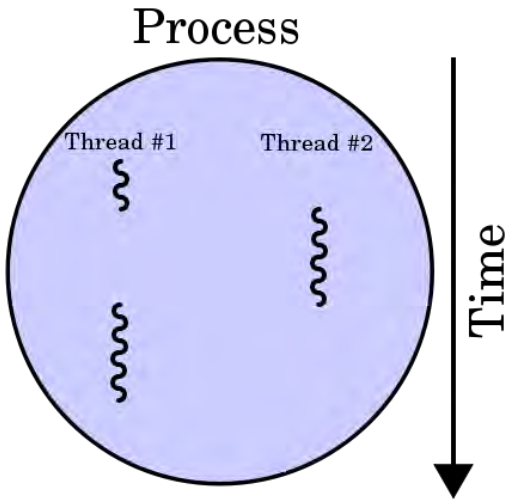


One possible architecture of a NUMA system. The processors connect to the bus or crossbar by connections of varying thickness/number. This shows that different CPUs have different access priorities to memory based on their relative location.

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

Threads

In computer science, a thread of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the operating system. [30] The implementation of threads and processes differs between operating systems, but in most cases a thread is a component of a process. Multiple threads can exist within one process, executing concurrently and sharing resources such as memory, while different processes do not share these resources. In particular, the threads of a process share its executable code and the values of its variables at any given time.



A process with two threads of execution, running on one processor

Threads made an early appearance in OS/360 Multiprogramming with a Variable Number of Tasks (MVT) in 1967, in which context they were called "tasks". The term "thread" has been attributed to Victor A. Vyssotsky. [31] Process schedulers of many modern operating systems directly support both time-sliced and multiprocessor threading, and the operating system kernel allows programmers to manipulate threads by exposing required functionality through the system call interface. Some threading implementations are called *kernel threads*, whereas *light-weight processes* (LWP) are a specific type of kernel thread that share the same state and information. Furthermore, programs can have *user-*

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

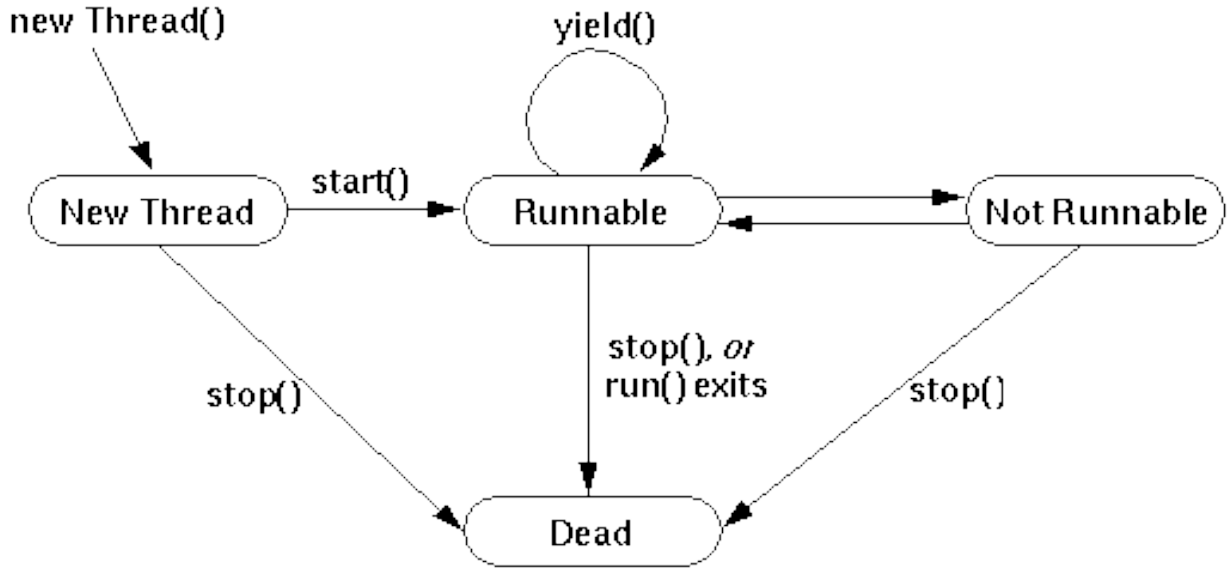
space threads when threading with timers, signals, or other methods to interrupt their own execution, performing a sort of *ad hoc* time slicing.

Threads vs. processes

Threads differ from traditional multitasking operating system processes in that:

- processes are typically independent, while threads exist as subsets of a process
- processes carry considerably more state information than threads, whereas multiple threads within a process share process state as well as memory and other resources
- processes have separate address spaces, whereas threads share their address space
- processes interact only through system-provided inter-process communication mechanisms
- context switching between threads in the same process is typically faster than context switching between processes.

Systems such as Windows NT and OS/2 are said to have *cheap* threads and *expensive* processes; in other operating systems there is not so great a difference except the cost of an address space switch which on some architectures (notably x86) results in a translation look aside buffer (TLB) flush.



Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

Single threading

In computer programming, *single-threading* is the processing of one command at a time. The opposite of single-threading is multithreading. While it has been suggested that the term *single-threading* is misleading, the term has been widely accepted within the functional programming community. With traditional single-threaded process implementation within a web server for example, the server can serve only one client request at a time and can make the waiting period for other users requesting services a very long time.

Multithreading

Multithreading is mainly found in multitasking operating systems. Multithreading is a widespread programming and execution model that allows multiple threads to exist within the context of one process. These threads share the process's resources, but are able to execute independently. The threaded programming model provides developers with a useful abstraction of concurrent execution. Multithreading can also be applied to one process to enable parallel execution on a multiprocessing system.

Multithreaded applications have the following advantages:

- *Responsiveness*: multithreading can allow an application to remain responsive to input. In a one-thread program, if the main execution thread blocks on a long-running task, the entire application can appear to freeze. By moving such long-running tasks to a *worker thread* that runs concurrently with the main execution thread, it is possible for the application to remain responsive to user input while executing tasks in the background. On the other hand, in most cases multithreading is not the only way to keep a program responsive, with non-blocking I/O and/or UNIX signals being available for gaining similar results. [32]
- *Faster execution*: this advantage of a multithreaded program allows it to operate faster on computer systems that have multiple central processing units (CPUs) or one or more multi-core processors, or across a cluster of machines, because the threads of the program naturally lend themselves to parallel execution, assuming sufficient independence (that they do not need to wait for each other).

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

- *Lower resource consumption*: using threads, an application can serve multiple clients concurrently using fewer resources than it would need when using multiple process copies of itself.
- *Better system utilization*: as an example, a file system using multiple threads can achieve higher throughput and lower latency since data in a faster medium (such as cache memory) can be retrieved by one thread while another thread retrieves data from a slower medium (such as external storage) with neither thread waiting for the other to finish.
- *Simplified sharing and communication*: unlike processes, which require a message passing or shared memory mechanism to perform inter-process communication (IPC), threads can communicate through data, code and files they already share.
- *Parallelization*: applications looking to use multicore or multi-CPU systems can use multithreading to split data and tasks into parallel subtasks and let the underlying architecture manage how the threads run, either concurrently on one core or in parallel on multiple cores. GPU computing environments like CUDA and OpenCL use the multithreading model where dozens to hundreds of threads run in parallel across data on a large number of cores.

Concurrency and data structures

Threads in the same process share the same address space. This allows concurrently running code to couple tightly and conveniently exchange data without the overhead or complexity of an IPC. When shared between threads, however, even simple data structures become prone to race conditions if they require more than one CPU instruction to update: two threads may end up attempting to update the data structure at the same time and find it unexpectedly changing underfoot. Bugs caused by race conditions can be very difficult to reproduce and isolate.

To prevent this, threading application programming interfaces (APIs) offer synchronization primitives such as mutexes to lock data structures against concurrent access. On uniprocessor systems, a thread running into a locked mutex must sleep and hence trigger a context switch. On multi-processor systems, the thread may instead poll the mutex in a spinlock. Both of these may sap performance and force processors in symmetric multiprocessing (SMP) systems to contend for the memory bus, especially if the granularity of the locking is fine.

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

Although threads seem to be a small step from sequential computation, in fact, they represent a huge step. They discard the most essential and appealing properties of sequential computation: understandability, predictability, and determinism. Threads, as a model of computation, are wildly non-deterministic, and the job of the programmer becomes one of pruning that non-determinism.

The Problem with Threads, Edward A. Lee, UC Berkeley, 2006

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

Challenges in Concurrent Programming

Race Conditions

A race condition occurs when two or more threads can access shared data and they try to change it at the same time. Because the thread scheduling algorithm can swap between threads at any time, you don't know the order in which the threads will attempt to access the shared data. Therefore, the result of the change in data is dependent on the thread scheduling algorithm, i.e. both threads are "racing" to access/change the data.

Problems often occur when one thread does a "check-then-act" (e.g. "check" if the value is X, then "act" to do something that depends on the value being X) and another thread does something to the value in between the "check" and the "act". E.g.:

```
if (x == 5) // The "Check"
{
   y = x * 2; // The "Act"

   // If another thread changed x in between "if (x == 5)" and "y = x * 2"
above,
   // y will not be equal to 10.
}
```

The point being, y could be 10, or it could be anything, depending on whether another thread changed x in between the check and act. You have no real way of knowing.

In order to prevent race conditions from occurring, you would typically put a lock around the shared data to ensure only one thread can access the data at a time. This would mean something like this: [40]

```
// Obtain lock for x
if (x == 5)
{
   y = x * 2; // Now, nothing can change x until the lock is released.
              // Therefore y = 10
}
// release lock for x
```


Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

Data Races

A data race occurs when two (or more) tasks attempt to access the same shared memory location, at least one of the accesses is a write, and the accesses may happen simultaneously.
For instance:

```
        int x = 0;
x = 1;  |||  printf("%d", x);
```

While race conditions may be benign, data races must be avoided! In many programming languages, they have very weak semantics (e.g., your program might crash).

Data races denote concurrent access to shared variables with insufficient lock protection, leading to a corrupted program state. Classical, or low-level, data races concern accesses to single fields. The notion of high-level data races deals with accesses to sets of related fields which should be accessed atomically. View consistency is a novel concept considering the association of variable sets to locks. This permits detecting high-level data races that can lead to an inconsistent program state, similar to classical low-level data races. Experiments on a small set of applications have shown that developers seem to follow the guideline of view consistency to a surprisingly large extent. Thus view consistency captures an important idea in multithreading design. [41]

Deadlocks

A deadlock is a state in which each member of a group is waiting for some other member to take action, such as sending a message or more commonly releasing a lock. [42] Deadlock is a common problem in multiprocessing systems, parallel computing, and distributed systems, where software and hardware locks are used to handle shared resources and implement process synchronization. [43]

In an operating system, a deadlock occurs when a process or thread enters a waiting state because a requested system resource is held by another waiting process, which in turn is waiting for another resource held by another waiting process. If a process is unable to change its state indefinitely because the resources requested by it are being used by another waiting process, then the system is said to be in a deadlock. [44]

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

In a communications system, deadlocks occur mainly due to lost or corrupt signals rather than resource contention. [45]

Necessary conditions

A deadlock situation on a resource can arise if and only if all of the following conditions hold simultaneously in a system: [44]

1. *Mutual exclusion:* The resources involved must be unshareable; otherwise, the processes would not be prevented from using the resource when necessary. Only one process can use the resource at any given instant of time.
2. *Hold and wait* or *resource holding:* a process is currently holding at least one resource and requesting additional resources which are being held by other processes.
3. *No preemption:* a resource can be released only voluntarily by the process holding it.
4. *Circular wait:* each process must be waiting for a resource which is being held by another process, which in turn is waiting for the first process to release the resource. In general, there is a set of waiting processes, $P = \{P_1, P_2, \ldots, P_N\}$, such that P_1 is waiting for a resource held by P_2, P_2 is waiting for a resource held by P_3 and so on until P_N is waiting for a resource held by P_1.

These four conditions are known as the *Coffman conditions* from their first description in a 1971 article by Edward G. Coffman, Jr.

Deadlock handling

Most current operating systems cannot prevent deadlocks. [44] When a deadlock occurs, different operating systems respond to them in different non-standard manners. Most approaches work by preventing one of the four Coffman conditions from occurring, especially the fourth one. Major approaches are as follows.

Ignoring deadlock

In this approach, it is assumed that a deadlock will never occur. This is also an application of the Ostrich algorithm. This approach was initially used by MINIX and UNIX. This is used when the time intervals between occurrences of deadlocks are large and the data loss incurred each time is tolerable.

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

Detection

Under the deadlock detection, deadlocks are allowed to occur. Then the state of the system is examined to detect that a deadlock has occurred and subsequently it is corrected. An algorithm is employed that tracks resource allocation and process states, it rolls back and restarts one or more of the processes in order to remove the detected deadlock. Detecting a deadlock that has already occurred is easily possible since the resources that each process has locked and/or currently requested are known to the resource scheduler of the operating system. [46]

After a deadlock is detected, it can be corrected by using one of the following methods:

1. *Process termination:* one or more processes involved in the deadlock may be aborted. One could choose to abort all competing processes involved in the deadlock. This ensures that deadlock is resolved with certainty and speed. But the expense is high as partial computations will be lost. Or, one could choose to abort one process at a time until the deadlock is resolved. This approach has high overhead because after each abort an algorithm must determine whether the system is still in deadlock. Several factors must be considered while choosing a candidate for termination, such as priority and age of the process.
2. *Resource preemption:* resources allocated to various processes may be successively preempted and allocated to other processes until the deadlock is broken.

Prevention

Deadlock prevention works by preventing one of the four Coffman conditions from occurring.

- Removing the *mutual exclusion* condition means that no process will have exclusive access to a resource. This proves impossible for resources that cannot be spooled. But even with spooled resources, deadlock could still occur. Algorithms that avoid mutual exclusion are called non-blocking synchronization algorithms.
- The *hold and wait* or *resource holding* conditions may be removed by requiring processes to request all the resources they will need before starting up (or

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

before embarking upon a particular set of operations). This advance knowledge is frequently difficult to satisfy and, in any case, is an inefficient use of resources. Another way is to require processes to request resources only when it has none. Thus, first they must release all their currently held resources before requesting all the resources they will need from scratch. This too is often impractical. It is so because resources may be allocated and remain unused for long periods. Also, a process requiring a popular resource may have to wait indefinitely, as such a resource may always be allocated to some process, resulting in resource starvation.[12] (These algorithms, such as serializing tokens, are known as the *all-or-none algorithms*.)

- The *no preemption* condition may also be difficult or impossible to avoid as a process has to be able to have a resource for a certain amount of time, or the processing outcome may be inconsistent or thrashing may occur. However, inability to enforce preemption may interfere with a *priority* algorithm. Preemption of a "locked out" resource generally implies a rollback, and is to be avoided, since it is very costly in overhead. Algorithms that allow preemption include lock-free and wait-free algorithms and optimistic concurrency control. If a process holding some resources and requests for some another resource(s) that cannot be immediately allocated to it, the condition may be removed by releasing all the currently being held resources of that process.
- The final condition is the *circular wait* condition. Approaches that avoid circular waits include disabling interrupts during critical sections and using a hierarchy to determine a partial ordering of resources. If no obvious hierarchy exists, even the memory address of resources has been used to determine ordering and resources are requested in the increasing order of the enumeration.[3] Dijkstra's solution can also be used.

Livelocks

A livelock is similar to a deadlock, except that the states of the processes involved in the livelock constantly change with regard to one another, none progressing. Livelock is a special case of resource starvation; the general definition only states that a specific process is not progressing.

A real-world example of livelock occurs when two people meet in a narrow corridor, and each tries to be polite by moving aside to let the other pass, but they end up swaying from side to side without making any progress because they both repeatedly move the same way at the same time.

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

Livelock is a risk with some algorithms that detect and recover from deadlock. If more than one process takes action, the deadlock detection algorithm can be repeatedly triggered. This can be avoided by ensuring that only one process (chosen randomly or by priority) takes action.

Resource Starvation

Starvation is a problem encountered in concurrent computing where a process is perpetually denied necessary resources to process its work. [47] Starvation may be caused by errors in a scheduling or mutual exclusion algorithm, but can also be caused by resource leaks, and can be intentionally caused via a denial-of-service attack such as a fork bomb.

The impossibility of starvation in a concurrent algorithm is called starvation-freedom, lockout-freedom[48] or finite bypass, [49] is an instance of liveness, and is one of the two requirements for any mutual exclusion algorithm (the other being correctness). The name "finite bypass" means that any process (concurrent part) of the algorithm is bypassed at most a finite number times before being allowed access to the shared resource. [49]

Starvation is usually caused by an overly simplistic scheduling algorithm. For example, if a (poorly designed) multi-tasking system always switches between the first two tasks while a third never gets to run, then the third task is being starved of CPU time. The scheduling algorithm, which is part of the kernel, is supposed to allocate resources equitably; that is, the algorithm should allocate resources so that no process perpetually lacks necessary resources.

Many operating system schedulers employ the concept of process priority. A high priority process A will run before a low priority process B. If the high priority process (process A) blocks and never yields, the low priority process (B) will (in some systems) never be scheduled—it will experience starvation. If there is an even higher priority process X, which is dependent on a result from process B, then process X might never finish, even though it is the most important process in the system. This condition is called a priority inversion. Modern scheduling algorithms normally contain code to guarantee that all processes will receive a minimum amount of each important resource (most often CPU time) in order to prevent any process from being subjected to starvation.

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

In computer networks, especially wireless networks, scheduling algorithms may suffer from scheduling starvation. An example is maximum throughput scheduling.

Starvation is similar to deadlock in that it causes a process to freeze. Two or more processes become deadlocked when each of them is doing nothing while waiting for a resource occupied by another program in the same set. On the other hand, a process is in starvation when it is waiting for a resource that is continuously given to other processes. Starvation-freedom is a stronger guarantee than the absence of deadlock: a mutual exclusion algorithm that must choose to let one of two processes into a critical section and picks one arbitrarily is deadlock-free, but not starvation-free. [49]

A possible solution to starvation is to use a scheduling algorithm with priority queue that also uses the aging technique. Aging is a technique of gradually increasing the priority of processes that wait in the system for a long time. [44]

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

Basic concepts and principles of concurrent programming

Atomicity - Linearizability

In concurrent programming, an operation (or set of operations) is atomic, linearizable, indivisible or uninterruptible if it appears to the rest of the system to occur at once without being interrupted. Atomicity is a guarantee of isolation from interrupts, signals, concurrent processes and threads. It is relevant for thread safety and reentrancy. Additionally, atomic operations commonly have a succeed-or-fail definition—they either successfully change the state of the system, or have no apparent effect.

In a concurrent system, processes can access a shared object at the same time. Because multiple processes are accessing a single object, there may arise a situation in which while one process is accessing the object, another process changes its contents. This example demonstrates the need for linearizability. In a linearizable system although operations overlap on a shared object, each operation appears to take place instantaneously. Linearizability is a strong correctness condition, which constrains what outputs are possible when an object is accessed by multiple processes concurrently. It is a safety property which ensures that operations do not complete in an unexpected or unpredictable manner. If a system is linearizable it allows a programmer to reason about the system. [12]

Atomicity is often enforced by mutual exclusion, whether at the hardware level building on a cache coherency protocol, or the software level using semaphores or locks. Thus, an atomic operation does not necessarily *actually* occur instantaneously. The benefit comes from the *appearance*: the system behaves *as if* each operation occurred instantly, separated by pauses. This makes the system consistent. Because of this, implementation details may be ignored by the user, except insofar as they affect performance. If an operation is not atomic, the user will also need to understand and cope with sporadic extraneous behavior caused by interactions between concurrent operations, which by their nature are likely to be hard to reproduce and debug.

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

Linearizability was first introduced as a consistency model by Herlihy and Wing in 1987. It encompassed more restrictive definitions of atomic, such as "an atomic operation is one which cannot be (or is not) interrupted by concurrent operations", which are usually vague about when an operation is considered to begin and end.

An atomic object can be understood immediately and completely from its sequential definition, as a set of operations run in parallel which always appear to occur one after the other; no inconsistencies may emerge. Specifically, linearizability guarantees that the invariants of a system are *observed* and *preserved* by all operations: if all operations individually preserve an invariant, the system as a whole will.

A concurrent system consists of a collection of processes communicating through shared data structures or objects. Linearizability is important in these concurrent systems where objects may be accessed by multiple processes at the same time and a programmer needs to be able to reason about the expected results. An execution of a concurrent system results in a *history*, an ordered sequence of completed operations.

A *history* is a sequence of *invocations* and *responses* made of an object by a set of threads or processes. An invocation can be thought of as the start of an operation, and the response being the signaled end of that operation. Each invocation of a function will have a subsequent response. This can be used to model any use of an object.

A *sequential* history is one in which all invocations have immediate responses, that is the invocation and response are considered to take place instantaneously. A sequential history should be trivial to reason about, as it has no real concurrency. This is where linearizability comes in.

A history σ is *linearizable* if there is a linear order of the completed operations such that:

1. For every completed operation in σ, the operation returns the same result in the execution as the operation would return if every operation was completed one by one in order σ.
2. If an operation op1 completes (gets a response) before op2 begins (invokes), then op1 precedes op2 in σ.[13]

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

In other words:

- its invocations and responses can be reordered to yield a sequential history;
- that sequential history is correct according to the sequential definition of the object;
- If a response preceded an invocation in the original history, it must still precede it in the sequential reordering.

Primitive atomic instructions

Processors have instructions that can be used to implement locking and lock-free and wait-free algorithms. The ability to temporarily inhibit interrupts, ensuring that the currently running process cannot be context switched, also suffices on a uniprocessor. These instructions are used directly by compiler and operating system writers but are also abstracted and exposed as bytecodes and library functions in higher-level languages:

- atomic read-write;
- atomic swap;
- test-and-set;
- fetch-and-add;
- compare-and-swap;
- Load-link / store-conditional.

High-level atomic operations

The easiest way to achieve linearizability is running groups of primitive operations in a critical section. Strictly, independent operations can then be carefully permitted to overlap their critical sections, provided this does not violate linearizability. Such an approach must balance the cost of large numbers of locks against the benefits of increased parallelism.

Another approach, favored by researchers (but not yet widely used in the software industry), is to design a linearizable object using the native atomic primitives provided by the hardware. This has the potential to maximize available parallelism and minimize synchronization costs, but requires mathematical proofs which show that the objects behave correctly.

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

A promising hybrid of these two is to provide a transactional memory abstraction. As with critical sections, the user marks sequential code that must be run in isolation from other threads. The implementation then ensures the code executes atomically. This style of abstraction is common when interacting with databases

A common theme when designing linearizable objects is to provide an all-or-nothing interface: either an operation succeeds completely, or it fails and does nothing. If the operation fails (usually due to concurrent operations), the user must retry, usually performing a different operation. For example:

- Compare-and-swap writes a new value into a location only if the latter's contents matches a supplied old value. This is commonly used in a read-modify-CAS sequence: the user reads the location, computes a new value to write, and writes it with a CAS (compare-and-swap); if the value changes concurrently, the CAS will fail and the user tries again.
- Load-link/store-conditional encodes this pattern more directly: the user reads the location with load-link, computes a new value to write, and writes it with store-conditional; if the value has changed concurrently, the SC (store-conditional) will fail and the user tries again.
- In a database transaction, if the transaction cannot be completed due to a concurrent operation (e.g. in a deadlock), the transaction will be aborted and the user must try again.

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

Sequential consistency

Sequential consistency is one of the consistency models used in the domain of concurrent computing (e.g. in distributed shared memory, distributed transactions, etc.).

It was first defined as the property that requires that

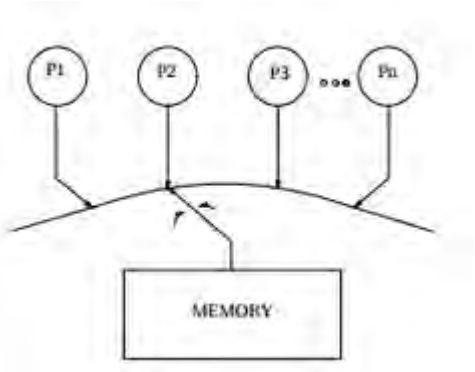
> **"... the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program."** [55]

To understand this statement, it is essential to understand one key property of sequential consistency: execution order of program in the same processor (or thread) is the same as the program order, while execution order of program between processors (or threads) is undefined. In an example like this:

```
processor 1:     <-- A1 run --> <-- B1 run -->        <-- C1 run -->
processor 2:       <-- A2 run --> <-- B2 run -->
Time ------------------------------------------------------------------------>
```

execution order between A1, B1 and C1 is preserved, that is, A1 runs before B1, and B1 before C1. The same for A2 and B2. But, as execution order between processors is undefined, B2 might run before or after C1 (B2 might physically run before C1, but the effect of B2 might be seen after that of C1, which is the same as "B2 run after C1")

Conceptually, there is single global memory and a "switch" that connects an arbitrary processor to memory at any time step. Each processor issues memory operations in program order and the switch provides the global serialization among all memory operations. [56]



Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

The sequential consistency is weaker than strict consistency, which requires a read from a location to return the value of the last write to that location; strict consistency demands that operations be seen in the order in which they were actually issued.

Data-race-free programming

As we mentioned before, data races denote concurrent access to shared variables with insufficient lock protection, leading to a corrupted program state. One more-or-less equivalent formulation of the above phrase can be produced based on total ordering of instructions. In more detail, **a** happens-before **b** in a program execution if

- **a** is sequenced before **b** in same thread
- **a** is a synchronization operation (e.g. lock release), that is observed by synchronization operation **b**.
- **a** happens-before **c** and **c** happens before **b**.

Consequently, a data race may be defined as conflicting accesses ordered by the happens-before principle. Thus, essentially data races are equal to non-determinism and undefined behavior making it impossible for someone to predict the future of an execution.

Data-race-free programming, meaning to produce pieces of concurrent and parallel code that do not contain data races, has become the programmer-centered model adopted to develop these pieces of code.

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

Transactional memory

Multi-core hardware is incredibly complex. In order to write concurrent/parallel programs we need to use abstractions. Abstraction is a way to introduce new concepts that are meaningful to humans. Abstraction tries to reduce and factor out details so that, e.g., the programmer can focus on a few concepts at a time.

Two common approaches are:

- Data parallelism
- Task-based parallelism

Data parallelism: the same operation is performed on different pieces of Data. On the advantages are:

+ Simple programming model
+ Convenient for certain numeric computations (e.g., matrix operations)
+ Parallelization (e.g., synchronization and load-balancing) can be left to the compiler and run-time system,

while a disadvantage is that it is not a universal programming model since it is only applicable to certain data structures and programming problems.

Task-based parallelism: operations are performed on separate threads that are coordinated with explicit synchronization (fork/join, locks, etc.)
This model places no restrictions on the code that each thread executes, when and how threads communicate, etc.

The main advantage is that it is a universal programming model which is capable of expressing all forms of parallel computation.
On the other hand, we should mention that it is a low level of abstraction, since it is close to hardware, and it is also very difficult to write correct programs.

Idea: Transactions provide a convenient abstraction also for coordinating reads and writes of shared data in a concurrent (or parallel) system. If we could wrap a computation in a transaction, we would get atomicity, consistency and isolation without having to worry about locking!
Since programs typically access shared data in memory, this approach to concurrency control is known as transactional memory.

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

In computer science and engineering, transactional memory attempts to simplify concurrent programming by allowing a group of load and store instructions to execute in an atomic way. It is a concurrency control mechanism analogous to database transactions for controlling access to shared memory in concurrent computing. Transactional memory systems provide high level abstraction as an alternative to low level thread synchronization. This abstraction allows for coordination between concurrent reads and writes of shared data in parallel systems. [22]

In concurrent programming, synchronization is required when parallel threads attempt to access a shared resource. Low level thread synchronization constructs such as locks are pessimistic and prohibit threads that are outside a critical section from making any changes. The process of applying and releasing locks often functions as additional overhead in workloads with little conflict among threads. Transactional memory provides optimistic concurrency control by allowing threads to run in parallel with minimal interference. The goal of transactional memory systems is to transparently support regions of code marked as transactions by enforcing atomicity, consistency, isolation and durability.

A transaction is a collection of operations that can execute and commit changes as long as a conflict is not present. When a conflict is detected, a transaction will revert to its initial state (prior to any changes) and will rerun until all conflicts are removed. Before a successful commit, the outcome of any operation is purely speculative inside a transaction. In contrast to lock-based synchronization where operations are serialized to prevent data corruption, transactions allow for additional parallelism as long as few operations attempt to modify a shared resource. Since the programmer is not responsible for explicitly identifying locks or the order in which they are acquired, programs that utilize transactional memory cannot produce a deadlock. [23]

With these constructs in place, transactional memory provides a high level programming abstraction by allowing programmers to enclose their methods within transactional blocks. Correct implementations ensure that data cannot be shared between threads without going through a transaction and produce a serializable outcome.

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

The concept of a transaction forms the foundation of transactional memory. Transactions first appeared as database unit abstractions with a defined set of attributes. Transactions must appear indivisible and instantaneous to the end-user or observer. The ACID properties were used as a requirement for database transactions, but apply equally to transactions for memory operations.

- **(A) Atomicity**

  Each transaction is atomic, and if part of the transaction fails then the entire transaction fails and the system state is left unchanged

- **(C) Consistency**

  Any transaction that is performed will take it from one consistent state to another. This is especially imperative when looking at transactional memory where the memory must remain in a consistent state while a transaction has taken place.

- **(I) Isolation**

  Other transactions or operations cannot access data that has been altered by a transaction currently in progress.

- **(D) Durability**

  Durability is described as the ability of a system to be able to recover committed transaction updates. In database systems this is especially important with regards to returning to a correct state after a system failure. In the case of transactional memory, it is the weakest requirement. Mainly, we can gather that once a transaction has succeeded, it cannot be lost. [24]

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

Advantages of transactional memory

- Easy to use synchronization construct
  - As easy to use as coarse-grain locks
  - Programmer declares, system implements
- Often performs as well as fine-grain locks
  - Automatic read-read concurrency & fine-grain concurrency
- Failure atomicity & recovery
  - No lost locks when a thread fails
  - Failure recovery = transaction abort + restart
- Composability
  - Safe & scalable composition of software modules

Atomic () ≠ lock () +unlock ()

- The difference
  - Atomic: high-level declaration of atomicity
    - Does not specify implementation/blocking behavior
    - Does note provide a consistency model
  - Lock: low-level blocking primitive
    - Does note provide atomicity or isolation on its own
- Keep in mind
  - Locks can be used to implement atomic (), but...
  - Locks can be used for purposes beyond atomicity
    - Cannot replace all lock regions with atomic regions
  - Atomic eliminates many data races, but...
  - Programming with atomic blocks can still suffer from atomicity violations. E.g., atomic sequence incorrectly split into two atomic blocks

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

Concurrent Data Structures

(e.g. *'Stacks'*, *'Queues'*, *'Pools'*, *'Linked lists'*, *'Hash Tables'*, *'Search Trees'*, *'Priority Queues'*)

In computer science, a concurrent data structure is a particular way of storing and organizing data for access by multiple computing threads (or processes) on a computer.

Historically, such data structures were used on uniprocessor machines with operating systems that supported multiple computing threads (or processes). The term concurrency captured the multiplexing/interleaving of the threads operations on the data by the operating system, even though the processors never issued two operations that accessed the data simultaneously.

Today, as multiprocessor computer architectures that provide parallelism become the dominant computing platform (through the proliferation of multi-core processors), the term has come to stand mainly for data structures that can be accessed by multiple threads which may actually access the data simultaneously because they run on different processors that communicate with one another. The concurrent data structure (sometimes also called a shared data structure) is usually considered to reside in an abstract storage environment called shared memory, though this memory may be physically implemented as either a "tightly coupled" or a distributed collection of storage modules. [14] [33]

Shared-memory multiprocessors are systems that concurrently execute multiple threads of computation which communicate and synchronize through data structures in shared memory. The efficiency of these data structures is crucial to *performance*. On today's machines, the layout of processors and memory, the layout of data in memory, the communication load on the various elements of the multiprocessor architecture all influence performance. Furthermore, the issues of correctness and performance are closely tied to each other: algorithmic enhancements that seek to improve performance often make it more difficult to design and verify a correct data structure implementation.

By most accounts, concurrent data structures are far more difficult to design than sequential ones because threads executing concurrently may interleave their steps in many ways, each with a different and potentially unexpected outcome. This requires designers to modify the way they think about computation, to

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

understand new design methodologies, and to adopt a new collection of programming tools. Furthermore, new challenges arise in designing *scalable* concurrent data structures that continue to perform well as machines that execute more and more concurrent threads become available) [33]

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

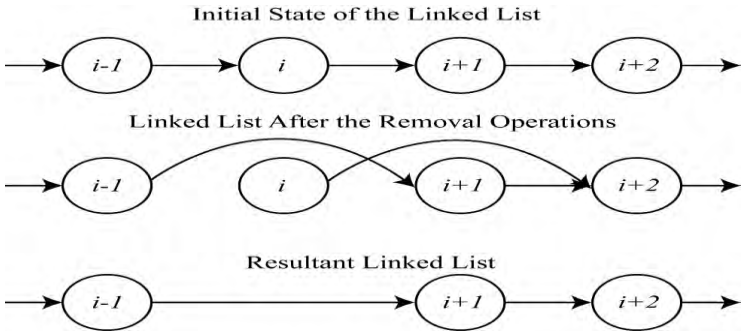
Preface on the Implementation Techniques

Mutual Exclusion

Mutual exclusion is a property of concurrency control, which is instituted for the purpose of preventing race conditions; it is the requirement that one thread of execution never enter its critical section at the same time that another concurrent thread of execution enters its own critical section.

The requirement of mutual exclusion was first identified and solved by Edsger W. Dijkstra in his seminal 1965 paper titled *Solution of a problem in concurrent programming control*, which is credited as the first topic in the study of concurrent algorithms. [34] [35]

A simple example of why mutual exclusion is important in practice can be visualized using a singly linked list of four items, where the second and third are to be removed. The removal of a node that sits between 2 other nodes is performed by changing the *next* pointer of the previous node to point to the next node (in other words, if node *i* is being removed, then the *next* pointer of node *i* − 1 is changed to point to node *i* + 1, thereby removing from the linked list any reference to node *i*). When such a linked list is being shared between multiple threads of execution, two threads of execution may attempt to remove two different nodes simultaneously, one thread of execution changing the *next* pointer of node *i* − 1 to point to node *i* + 1, while another thread of execution changes the *next* pointer of node *i* to point to node *i* + 2. Although both removal operations complete successfully, the desired state of the linked list is not achieved: node *i* + 1 remains in the list, because the *next* pointer of node *i* − 1 points to node *i* + 1.



Two nodes, *i* and *i* + 1, being removed simultaneously results in node *i* + 1 not being removed.

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

This problem (called a *race condition*) can be avoided by using the requirement of mutual exclusion to ensure that simultaneous updates to the same part of the list cannot occur.

The term mutual exclusion is also used in reference to the simultaneous writing of a memory address by one thread while the aforementioned memory address is being manipulated or read by another thread or other threads.

The problem which mutual exclusion addresses is a problem of resource sharing: how can a software system control multiple processes' access to a shared resource, when each process needs exclusive control of that resource while doing its work? The mutual-exclusion solution to this makes the shared resource available only while the process is in a specific code segment called the critical section. It controls access to the shared resource by controlling each mutual execution of that part of its program where the resource would be used.

A successful solution to this problem must have at least these two properties:

- It must implement *mutual exclusion*: only one process can be in the critical section at a time.
- It must be free of *deadlocks*: if processes are trying to enter the critical section, one of them must eventually be able to do so successfully, provided no process stays in the critical section permanently.

There exist both software and hardware solutions for enforcing mutual exclusion. Some different solutions are discussed below.

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

Hardware solutions

On uniprocessor systems, the simplest solution to achieve mutual exclusion is to disable interrupts during a process's critical section. This will prevent any interrupt service routines from running (effectively preventing a process from being preempted). Although this solution is effective, it leads to many problems. If a critical section is long, then the system clock will drift every time a critical section is executed because the timer interrupt is no longer serviced, so tracking time is impossible during the critical section. Also, if a process halts during its critical section, control will never be returned to another process, effectively halting the entire system. A more elegant method for achieving mutual exclusion is the busy-wait.

Busy-waiting is effective for both uniprocessor and multiprocessor systems. The use of shared memory and an atomic test-and-set instruction provide the mutual exclusion. A process can test-and-set on a location in shared memory, and since the operation is atomic, only one process can set the flag at a time. Any process that is unsuccessful in setting the flag can either go on to do other tasks and try again later, release the processor to another process and try again later, or continue to loop while checking the flag until it is successful in acquiring it. Preemption is still possible, so this method allows the system to continue to function—even if a process halts while holding the lock.

Several other atomic operations can be used to provide mutual exclusion of data structures; most notable of these is compare-and-swap (CAS). CAS can be used to achieve wait-free mutual exclusion for any shared data structure by creating a linked list where each node represents the desired operation to be performed. CAS is then used to change the pointers in the linked list [36] during the insertion of a new node. Only one process can be successful in its CAS; all other processes attempting to add a node at the same time will have to try again. Each process can then keep a local copy of the data structure, and upon traversing the linked list, can perform each operation from the list on its local copy.

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

Software solutions

Beside hardware-supported solutions, some software solutions exist that use busy waiting to achieve mutual exclusion. Examples of these include the following:

- Dekker's algorithm
- Peterson's algorithm
- Lamport's bakery algorithm
- Szymanski's algorithm
- Taubenfeld's black-white bakery algorithm.

These algorithms do not work if out-of-order execution is used on the platform that executes them. Programmers have to specify strict ordering on the memory operations within a thread. [37]

It is often preferable to use synchronization facilities provided by an operating system's multithreading library, which will take advantage of hardware solutions if possible but will use software solutions if no hardware solutions exist.

The solutions explained above can be used to build the synchronization primitives below:

- locks (mutexes);
- readers–writer locks
- recursive locks
- semaphores
- monitors
- message passing
- Tuple space.

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

Locks

In computer science, a lock or mutex (from mutual exclusion) is a synchronization mechanism for enforcing limits on access to a resource in an environment where there are many threads of execution. A lock is designed to enforce a mutual exclusion concurrency control policy. [14]

Generally, locks are *advisory locks*, where each thread cooperates by acquiring the lock before accessing the corresponding data. Some systems also implement *mandatory locks*, where attempting unauthorized access to a locked resource will force an exception in the entity attempting to make the access.

The simplest type of lock is a binary semaphore. It provides exclusive access to the locked data. Other schemes also provide shared access for reading data. Other widely implemented access modes are exclusive, intend-to-exclude and intend-to-upgrade.

Another way to classify locks is by what happens when the lock strategy prevents progress of a thread. Most locking designs block the execution of the thread requesting the lock until it is allowed to access the locked resource. With a spinlock, the thread simply waits ("spins") until the lock becomes available. This is efficient if threads are blocked for a short time, because it avoids the overhead of operating system process re-scheduling. It is inefficient if the lock is held for a long time, or if the progress of the thread that is holding the lock depends on preemption of the locked thread.

Locks typically require hardware support for efficient implementation. This support usually takes the form of one or more atomic instructions such as "test-and-set", "fetch-and-add" or "compare-and-swap". These instructions allow a single process to test if the lock is free, and if free, acquire the lock in a single atomic operation.

Uniprocessor architectures have the option of using uninterruptable sequences of instructions—using special instructions or instruction prefixes to disable interrupts temporarily—but this technique does not work for multiprocessor shared-memory machines. Proper support for locks in a multiprocessor environment can require quite complex hardware or software support, with substantial synchronization issues.

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

The reason an atomic operation is required is because of concurrency, where more than one task executes the same logic. For example, consider the following C code:

```
if(lock == 0) {
    // lock free, set it
    lock = myPID;
}
```

The above example does not guarantee that the task has the lock, since more than one task can be testing the lock at the same time. Since both tasks will detect that the lock is free, both tasks will attempt to set the lock, not knowing that the other task is also setting the lock. Dekker's or Peterson's algorithm are possible substitutes if atomic locking operations are not available.

Careless use of locks can result in deadlock or livelock. A number of strategies can be used to avoid or recover from deadlocks or livelocks, both at design-time and at run-time. (The most common strategy is to standardize the lock acquisition sequences so that combinations of inter-dependent locks are always acquired in a specifically defined "cascade" order.)

Disadvantages

Lock-based resource protection and thread/process synchronization have many disadvantages:

- Contention: some threads/processes have to wait until a lock (or a whole set of locks) is released. If one of the threads holding a lock dies, stalls, blocks, or enters an infinite loop, other threads waiting for the lock may wait forever.
- Overhead: the use of locks adds overhead for each access to a resource, even when the chances for collision are very rare. (However, any chance for such collisions is a race condition.)
- Debugging: bugs associated with locks are time dependent and can be very subtle and extremely hard to replicate, such as deadlocks.
- Instability: the optimal balance between lock overhead and lock contention can be unique to the problem domain (application) and sensitive to design, implementation, and even low-level system architectural changes. These balances may change over the life cycle of an application and may entail tremendous changes to update (re-balance).

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

- Composability: locks are only composable (e.g., managing multiple concurrent locks in order to atomically delete item X from table A and insert X into table B) with relatively elaborate (overhead) software support and perfect adherence by applications programming to rigorous conventions.
- Priority inversion: a low-priority thread/process holding a common lock can prevent high-priority threads/processes from proceeding. Priority inheritance can be used to reduce priority-inversion duration. The priority ceiling protocol can be used on uniprocessor systems to minimize the worst-case priority-inversion duration, as well as prevent deadlock.
- Convoying: all other threads have to wait if a thread holding a lock is descheduled due to a time-slice interrupt or page fault.

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

Spinlocks

A spinlock is a lock which causes a thread trying to acquire it to simply wait in a loop ("spin") while repeatedly checking if the lock is available. Since the thread remains active but is not performing a useful task, the use of such a lock is a kind of busy waiting. Once acquired, spinlocks will usually be held until they are explicitly released, although in some implementations they may be automatically released if the thread being waited on (that which holds the lock) blocks, or "goes to sleep".

Because they avoid overhead from operating system process rescheduling or context switching, spinlocks are efficient if threads are likely to be blocked for only short periods. For this reason, operating-system kernels often use spinlocks. However, spinlocks become wasteful if held for longer durations, as they may prevent other threads from running and require rescheduling. The longer a thread holds a lock, the greater the risk that the thread will be interrupted by the OS scheduler while holding the lock. If this happens, other threads will be left "spinning" (repeatedly trying to acquire the lock), while the thread holding the lock is not making progress towards releasing it. The result is an indefinite postponement until the thread holding the lock can finish and release it. This is especially true on a single-processor system, where each waiting thread of the same priority is likely to waste its quantum (allocated time where a thread can run) spinning until the thread that holds the lock is finally finished.

Implementing spin locks correctly offers challenges because programmers must take into account the possibility of simultaneous access to the lock, which could cause race conditions. Generally, such implementation is possible only with special assembly-language instructions, such as atomic test-and-set operations, and cannot be easily implemented in programming languages not supporting truly atomic operations. [38] On architectures without such operations, or if high-level language implementation is required, a non-atomic locking algorithm may be used, e.g. Peterson's algorithm. But note that such an implementation may require more memory than a spinlock, be slower to allow progress after unlocking, and may not be implementable in a high-level language if out-of-order execution is allowed.

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

Lock-free

An algorithm is called non-blocking if failure or suspension of any thread cannot cause failure or suspension of another thread for some operations. These algorithms provide a useful alternative to traditional blocking implementations. A non-blocking algorithm is lock-free if there is guaranteed system-wide progress, and wait-free if there is also guaranteed per-thread progress.

The word "non-blocking" was traditionally used to describe telecommunications networks that could route a connection through a set of relays "without having to re-arrange existing calls". Also, if the telephone exchange "is not defective, it can always make the connection".

The traditional approach to multi-threaded programming is to use locks to synchronize access to shared resources. Synchronization primitives such as mutexes, semaphores, and critical sections are all mechanisms by which a programmer can ensure that certain sections of code do not execute concurrently, if doing so would corrupt shared memory structures. If one thread attempts to acquire a lock that is already held by another thread, the thread will block until the lock is free.

Blocking a thread can be undesirable for many reasons. An obvious reason is that while the thread is blocked, it cannot accomplish anything: if the blocked thread had been performing a high-priority or real-time task, it would be highly undesirable to halt its progress.

Other problems are less obvious. For example, certain interactions between locks can lead to error conditions such as deadlock, livelock, and priority inversion. Using locks also involves a trade-off between coarse-grained locking, which can significantly reduce opportunities for parallelism, and fine-grained locking, which requires more careful design, increases locking overhead and is more prone to bugs.

Unlike blocking algorithms, non-blocking algorithms do not suffer from these downsides, and in addition are safe for use in interrupt handlers: even though the preempted thread cannot be resumed, progress is still possible without it. In contrast, global data structures protected by mutual exclusion cannot safely be accessed in an interrupt handler, as the preempted thread may be the one holding the lock.

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

A lock-free data structure can be used to improve performance. A lock-free data structure increases the amount of time spent in parallel execution rather than serial execution, improving performance on a multi-core processor, because access to the shared data structure does not need to be serialized to stay coherent.

With few exceptions, non-blocking algorithms use atomic read-modify-write primitives that the hardware must provide, the most notable of which is compare and swap (CAS). Critical sections are almost always implemented using standard interfaces over these primitives. Until recently, all non-blocking algorithms had to be written "natively" with the underlying primitives to achieve acceptable performance. However, the emerging field of software transactional memory promises standard abstractions for writing efficient non-blocking code.

Much research has also been done in providing basic data structures such as stacks, queues, sets, and hash tables. These allow programs to easily exchange data between threads asynchronously.

Additionally, some non-blocking data structures are weak enough to be implemented without special atomic primitives. These exceptions include:

- a single-reader single-writer ring buffer FIFO, with a size which evenly divides the overflow of one of the available unsigned integer types, can unconditionally be implemented safely using only a memory barrier
- Read-copy-update with a single writer and any number of readers.
- Read-copy-update with multiple writers and any number of readers.

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

Read Copy Update (RCU)

In computer science, read-copy-update (RCU) is a synchronization mechanism, that was added to the Linux kernel in October of 2002, based on mutual exclusion. [39] It is used when performance of reads is crucial and is an example of space–time tradeoff, enabling fast operations at the cost of more space.

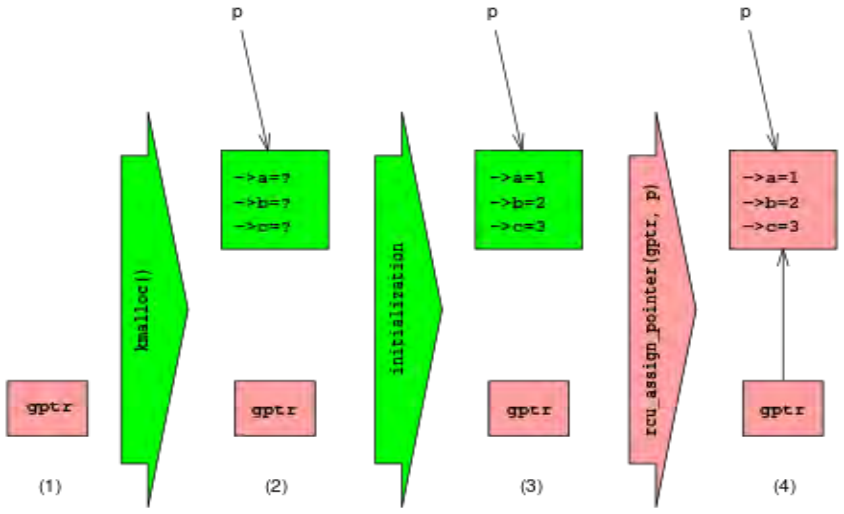
Read-copy-update allows multiple threads to efficiently read from shared memory by deferring updates after pre-existing reads to a later time while simultaneously marking the data, ensuring new readers will read the updated data. This makes all readers proceed as if there were no synchronization involved, hence they will be fast, but also making updates more difficult.

RCU achieves scalability improvements by allowing reads to occur concurrently with updates. In contrast with conventional locking primitives that ensure mutual exclusion among concurrent threads regardless of whether they be readers or updaters, or with reader-writer locks that allow concurrent reads but not in the presence of updates, RCU supports concurrency between a single updater and multiple readers. RCU ensures that reads are coherent by maintaining multiple versions of objects and ensuring that they are not freed up until all pre-existing read-side critical sections complete. RCU defines and uses efficient and scalable mechanisms for publishing and reading new versions of an object, and also for deferring the collection of old versions. These mechanisms distribute the work among read and update paths in such a way as to make read paths extremely fast. In some cases (non-preemptable kernels), RCU's read-side primitives have zero overhead.

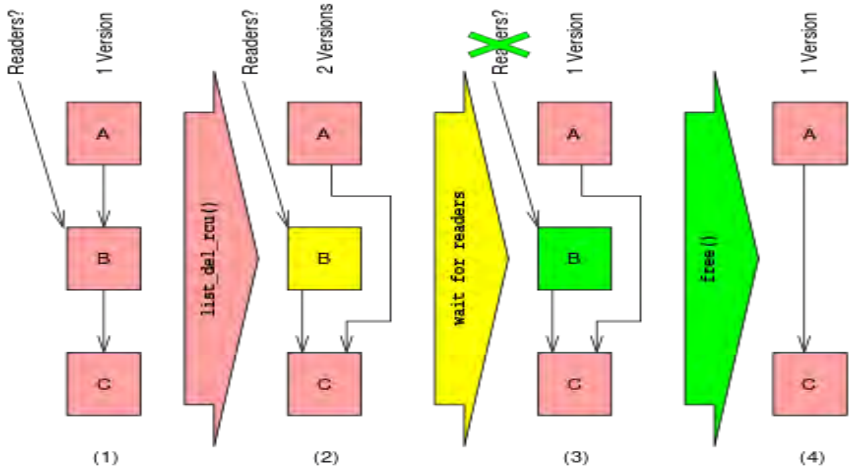
Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

The typical RCU update sequence goes something like the following:

1. Ensure that all readers accessing RCU-protected data structures carry out their references from within an RCU read-side critical section.
2. Remove pointers to a data structure, so that subsequent readers cannot gain a reference to it.
3. Wait for a grace period to elapse, so that all previous readers (which might still have pointers to the data structure removed in the prior step) will have completed their RCU read-side critical sections.
4. At this point, there cannot be any readers still holding references to the data structure, so it now may safely be reclaimed (e.g., freed).



Read-copy-update insertion procedure. A thread allocates a structure with three fields, then sets the global pointer gptr to point to this structure.



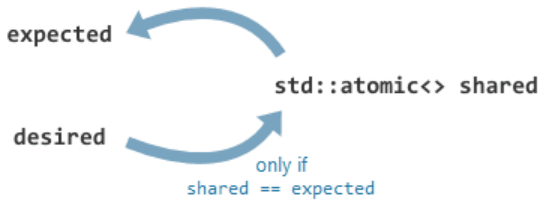
Read-copy-update deletion procedure

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

In the above procedure (which matches the earlier diagram), the updater is performing both the removal and the reclamation step, but it is often helpful for an entirely different thread to do the reclamation. Reference counting can be used to let the reader perform removal so, even if the same thread performs both the update step (step (2) above) and the reclamation step (step (4) above), it is often helpful to think of them separately.

Compare and Swap

Compare-and-swap (CAS) is an atomic instruction used in multithreading to achieve synchronization. It compares the contents of a memory location with a given value and, only if they are the same, modifies the contents of that memory location to a new given value. This is done as a single atomic operation. The atomicity guarantees that the new value is calculated based on up-to-date information; if the value had been updated by another thread in the meantime, the write would fail. The result of the operation must indicate whether it performed the substitution; this can be done either with a simple Boolean response (this variant is often called compare-and-set), or by returning the value read from the memory location (*not* the value written to it).



Compare-and-swap (and compare-and-swap-double) has been an integral part of the IBM 370 (and all successor) architectures since 1970. The operating systems that run on these architectures make extensive use of this instruction to facilitate process (i.e., system and user tasks) and processor (i.e., central processors) parallelism while eliminating, to the greatest degree possible, the "disabled spin locks" which had been employed in earlier IBM operating systems. Similarly, the use of test-and-set was also eliminated. In these operating systems, new units of work may be instantiated "globally", into the global service priority list, or "locally", into the local service priority list, by the execution of a single compare-and-swap instruction. This substantially improved the responsiveness of these operating systems.

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

Implementation Part

Experimentation Setup

This thesis is determined to not only explore the theoretical aspect of these matters, but also the practical part as well. Indeed, the uttermost goal is to evaluate the behavior of sorted lists, accessed in a multithreaded fashion, in terms of speed, scalability, suitability and ease of implementation.

This evaluation will be conducted with respect to the prototypes explained above. That means a variation of techniques that make a list suitable for concurrent accesses. This includes the coarse and fine-grained locking of the nodes using simple mutex locks, implementation of a Test and Test and Set spinlock and applying it instead of the mutex ones, using the lock-free compare and Swap technique and finally using gcc's library of transactional memory.

Furthermore, a simple benchmarking library is developed In order to correctly measure the performance of each one of them. All these were developed in C++ and compiled according to the g++11 standard.

Besides the details regarding purely the implementation part, a discussion section will follow up, containing the outcomes of the benchmarking.

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

Of course, we have to gain some intelligence about the hardware that we are going to experiment on, since this is a critical matter for our benchmarking. After running the lscpu command on the terminal, we gained the following information about our system.

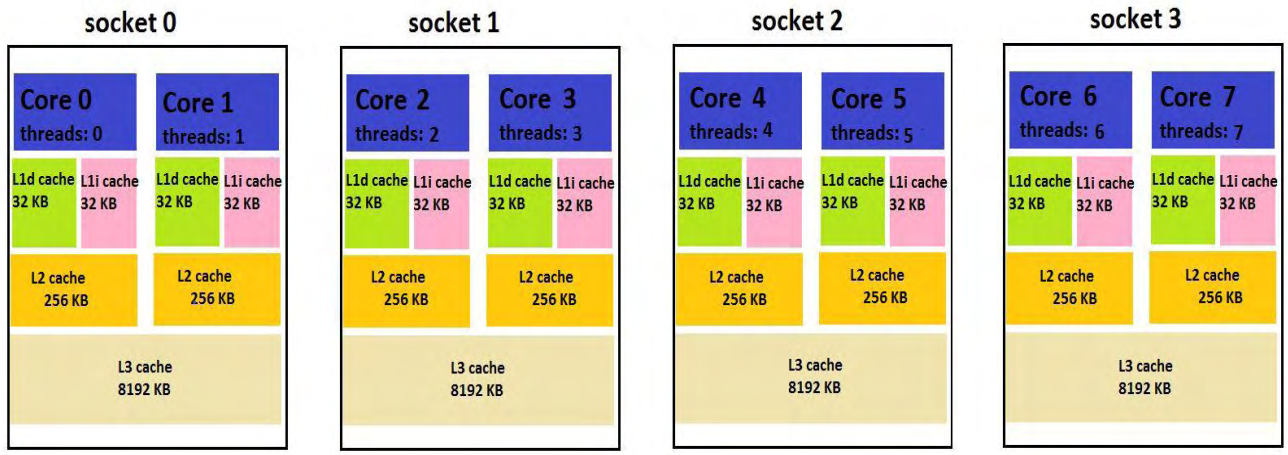
```
lepa@ubuntu:~$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                8
On-line CPU(s) list:   0-7
Thread(s) per core:    1
Core(s) per socket:    2
Socket(s):             4
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 60
Model name:            Intel(R) Core(TM) i7-4790K CPU @ 4.00GHz
Stepping:              3
CPU MHz:               3997.683
BogoMIPS:              7995.36
Hypervisor vendor:     VMware
Virtualization type:   full
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              8192K
NUMA node0 CPU(s):     0-7
```

When we trigger the –p flag, the detailed map of the multicore architecture can be seen.

```
lepa@ubuntu:~$ lscpu -p
# The following is the parsable format, which can be fed to other
# programs. Each different item in every column has an unique ID
# starting from zero.
# CPU,Core,Socket,Node,,L1d,L1i,L2,L3
0,0,0,0,,0,0,0,0
1,1,0,0,,0,0,0,0
2,2,1,0,,1,1,1,1
3,3,1,0,,1,1,1,1
4,4,2,0,,2,2,2,2
5,5,2,0,,2,2,2,2
6,6,3,0,,3,3,3,3
7,7,3,0,,3,3,3,3
```


Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

Which translates to:



Detailed description of the Data Structure and available functions in the API

For the first mutex test, we implement a sorted simply-linked list with no sentinels containing three (3) basic functions in its API. The implementation of the list is simple enough. Every node consists only of its value and a pointer to the next node. The first function is the "insert" which scans the existing list, finds the proper position of the newly constructed node, and inserts it in the list. It represents the "write" function. The second function is the "remove" which, given a node, either finds the node in the list and removes it, or fails and prints the corresponding message. It represents another type of "write" function. The third and final function is "count" which counts the number of nodes containing the number given as input which is the same as a "read" function. In more detail:

Insert(int v): When this function is called, firstly, traverses through the list and searches for the position to add the given integer. When found, it allocates space for the new node, copies the value inside it, and places it in the position found before changing the corresponding pointers. This way, our list is being sorted the whole time.

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

Remove(int v): When this function is called, firstly, traverses through the list and searches if the input number is an element of the list. If not, then it returns and goes out of scope. If it is then it changes the corresponding pointers to bypass this node, thus deleting it virtually, and then it frees the node, deleting it physically too.

Count(int v): When this function is called, traverses through the list searching for the given number. When found we augment a counter dedicated to counting the number of appearances of the specific number. When it finishes, it falls out of scope returning this counter.

These very same functions and this data structure is also used for benchmarking the spinlock implementation too.

As far as the lock free implementation is concerned, the only thing we changed is that we added a sentinel to the list which at that timepoint seemed to deal better with extreme situations as removing the head of the list and so on. This is something that doesn't really affect our measurements significantly in any way.

Finally, the transactional memory implementation the concept involved implementing a doubly-linked queue with right and left sentinels, and a head pointer pointing at them. Also, the functions available in this benchmarking, were PushLeft, PushRight, PopLeft and PopRight representing browsing shared memory in a "write" manner, inserting and deleting.

PushLeft: When this function is called, a new node of the Queue with its value is created. Then the new node's right is set to the old right and the old right's left is set to the new node. Respectively the new node's left is set to sentinel and the sentinel's right is set to the new node.

PushRight: Similarly with the PushLeft, when this function is called, a new node of the Queue with its value is created. Then the new node's left is set to the old left and the old left's right is set to the new node. Respectively the new node's right is set to sentinel and the sentinel's left is set to the new node.

PopLeft: When this function is called, the left node of the list is deleted. This is accomplished by pointing the sentinel to the 2nd node from the left and deleting the 1st node by pointing its right to the left sentinel.

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

PopRight: Similarly, with the PopLeft, when this function is called, the right node of the list is deleted. This is accomplished by pointing the sentinel to the 2nd node from the right and deleting the last node by pointing its left to the right sentinel.

Adaptation to the theory

Below follows a detailed description of the aforementioned concurrent techniques. That should be coarse and fine-grained locking of the list, further experimentation of gcc's memory library by implementing our own spinlock (TestAndTestAndSet style) and integration in the above motifs, lock-free, and using transactional memory library in gcc.

We initially developed the "**Coarse grain**" and "**Fine grain**" locking scheme of the list. The first one (Coarse grain) seemed to be the most straight forward. Each time a method was called, we had to lock the whole structure. So, we declared a mutex lock, and when insert, delete and count was called, we locked the mutex, so that every other thread that tried to access its method could not do that. So, this meant that e.g. when inserting or deleting or counting something from the list, other threads had to wait for the lock to be unlocked, in order to execute the function.

HUGE difference between these two is that the second requires a lock integrated into every node!!

The fine-grained solution seemed to be more conceptually correct regarding the concurrent accesses on the list. However, we can't implement the fine grain locking with only one lock. So, having two locks every time a thread wants to access a node of the list, we make sure that no other thread can "touch" the place we want to change. As reasonable as the design may seem though, the implementation was not that easy at all. We had to add a mutex inside every node of the struct and another one for locking the head, taking care of the scenarios messing with the head of the list. Of course, the remove method was much more difficult to take care of than the insert or count methods. Furthermore, concerning the counting method we had to add a lock specifically for the first node, just to take care of the edge case that we have to count the number of appearances of the first value of the list.

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

TATAS LOCK -- COARSE-GRAINED TATAS -- FINE-GRAINED TATAS

The second part contains the implementation of our own locking technique using the Test And Test And Set method. The list remains the same (sorted simply-linked list with no sentinels) with the same 3 functions insert, remove and count.

Instead of the previous mutex we created a spinlock using std:atomic type. In the lock() method, a do-while loop is used with another while loop inside the block.

Test part: The inner while loop loads from the value stored in the atomic Boolean and compares it to “false” as “false” means the lock is free. If the returned value is “true”, it means the lock is still in use by another thread.

Test and set part: When the inner loop is broken, the condition of the outer do-while loop is processed, which writes to the atomic Boolean the value “true”, meaning it will be locked. The exchange () method returns the previous value of the Boolean, so it is compared with “false” to ensure that it was our thread that got the lock. If the comparison succeeds, it means another thread got the lock and the inner while loop starts executing again. If the comparison fails, it means we got the lock so the outer loop breaks as well.

How is it in fact implemented in C++?

The flag variable has to be atomic and thus we create and atomic bool type variable. As a result, in order to handle this variable, we have to use the corresponding atomic load and store directives. A memory prototype has to be followed and it is chosen from the one in the std::memory_order library.

Memory order relaxed: Relaxed operation: there are no synchronization or ordering constraints imposed on other reads or writes, only this operation's atomicity is guaranteed

Memory order acquire: A load operation with this memory order performs the acquire operation on the affected memory location: no reads or writes in the current thread can be reordered before this load. All writes in other threads that release the same atomic variable are visible in the current thread

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

Memory order release: A store operation with this memory order performs the release operation: no reads or writes in the current thread can be reordered after this store. All writes in the current thread are visible in other threads that acquire the same atomic variable and writes that carry a dependency into the atomic variable become visible in other threads that consume the same atomic

Load: Atomically loads and returns the current value of the atomic variable. Memory is affected according to the value of order.

Store: Atomically replaces the current value with desired. Memory is affected according to the value of order.

After that, the only thing that we had to do, was to change our first coarse and fine grain lock implementation of the list. That means that we had to replace our mutexes used, with our TATAS lock. The rest remained the same in terms of logic behind locking the concurrent data structure.

LOCK FREE IMPLEMENTATION

The lock-free solution was the most intriguing of them all. First of all, some words about implementing the list. We almost kept the same structure with some changes in respect to our goal. The pointer inside the structure now has to become a shared pointer as well as the one pointing to our head. Our list in this case has a sentinel, in order to have less cases to take care of when inserting and deleting. Regarding the insert and delete functions, now everything works “a bit” differently. All pointers used are shared pointers and the reason for that is what was stated in the instructions, garbage collection. Also, we have implemented a copy_list method whose work is to just take as argument a pointer of the list and create a new one, identical, returning us the head of the new copied list. Consider the case of a thread intending to insert or delete something on the list. It makes a copy of the list while keeping the head of the old list, and it proceeds inserting or deleting whatever it wants on its own copied list not having to do anything at all with the original list.

Afterwards, when it is done modifying its own list, it checks to see if the head of the original list that had kept in the beginning is the same up until now - meaning that no one else had modified the list in the meantime- and if this is true it swaps

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

the head of the original list to the head of its list, making its list the original one. This is achieved via atomic_compare_exhange guaranteeing us atomicity in the compare and swap. If someone else had changed the list in the meantime and the CAS instruction fails, it proceeds taking the new head and starting over, and that is the reason our insert and delete function are wrapped up in a while(true) loop. We have also to check if the list has changed even if a thread wanted to delete something and did not find it, because it may have been just inserted for example. Of course, as aforementioned, the copied lists that should have been freed after every iteration or after completed CAS are freed automatically as we have used shared pointers. Another thing that we have to note at this point, is that we kept getting data races as long as we just assigned addresses to the pointers, and not atomically load them. Keep in mind that this kind of structure doesn't have to take special care of count- or else reading- as changes made from every thread are made on their private copies.

TRANSACTIONAL MEMORY

For the last implementation we developed a queue doubly-linked with two sentinels, head of which point at them. Available functions in its API, as mentioned above, are 4 write function. The main idea as discussed, is the ease of implementation of the programmer, translating into physical hardware and guaranteeing Atomicity, Consistency, Isolation, Durability properties on the critical region. This essentially means that wrap around every critical region – shared memory, with an atomic transaction, thus ensuring correct concurrent properties.

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

Detailed description of the benchmarking structure

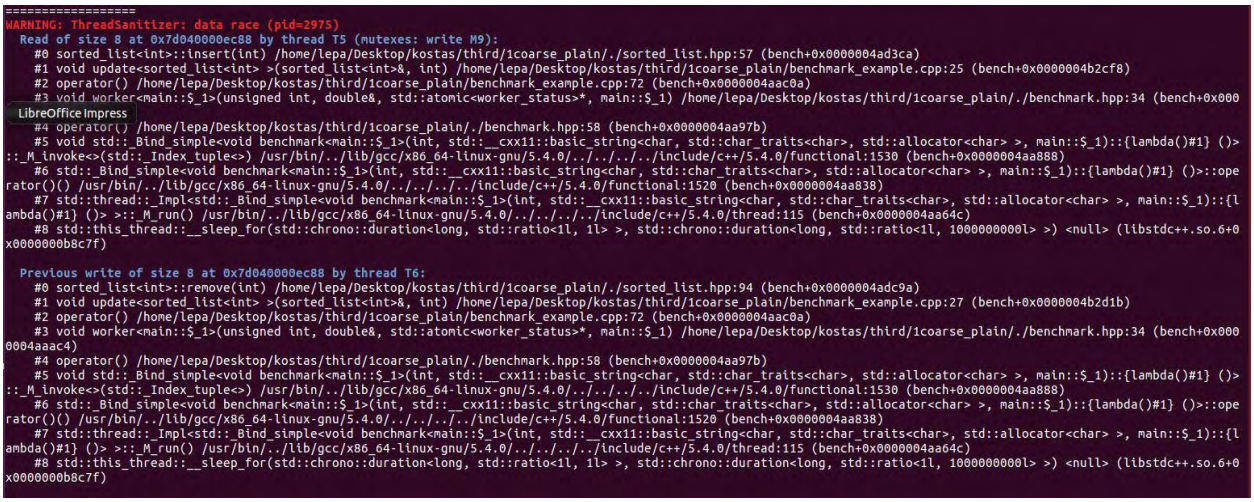
Main concern of the benchmarking part was to produce legitimate and totally unbiased results and there were a few assumptions made to help us move towards that target. Firstly, before benchmarking starts, we prefill our list with numbers, integers as we chose. Secondly, we wait for all worker threads to start working and then start counting the time, so creation overhead is not included. More importantly we run each benchmark for a specific amount of seconds and then divide by this number to get a mean value of operations, so no noticeable differentiation on the results could happen. Besides that, we run 3 different benchmarks each one containing different combination of read and write operations on the list, one containing only read accesses referred as Read, one containing only write operations, referred as Update and the other one containing both, referred as Mixed. The result is returned and counted in thousands of operations per second (kops/sec).

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

Debugging issues

This section is devoted to the debugging issues that we faced during the development of this thesis. It is considered an equal part of the process, since data-race hunting was one of the most time consuming processes. Embedded into pretty much every IDE one can find the thread sanitizer tool, base of whom is the native thread sanitizer tool provided from gcc.

The most common data race we faced, was during the development of the fine-grained locking mechanism from the list, something totally reasonable if we consider the amount locks we had to deal with.



Example of a data race. Feedback with exact trace of the crash by gcc's thread sanitizer.

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

Analysis of the trace

In the previous screenshot we can see a data race example that we faced during the Coarse-grained implementation of the single-linked list. It concerned the case where we had not locked the remove function (line 76). Running, during our compile, the –fsanitizer –g command, we are provided by a complete path of the code that is executed, the data races that happen, even their exact position. The data race occurred when 2 different threads (T5 & T6) tried to access the same place in memory in a different manner. As we can see, the exact same time that T6 "writes" at 0x7d040000ec88, T5 is trying to read from the same address. Inevitably a data race happens since there is an obvious conflict about the correct value of the 0x7d040000ec88 address.

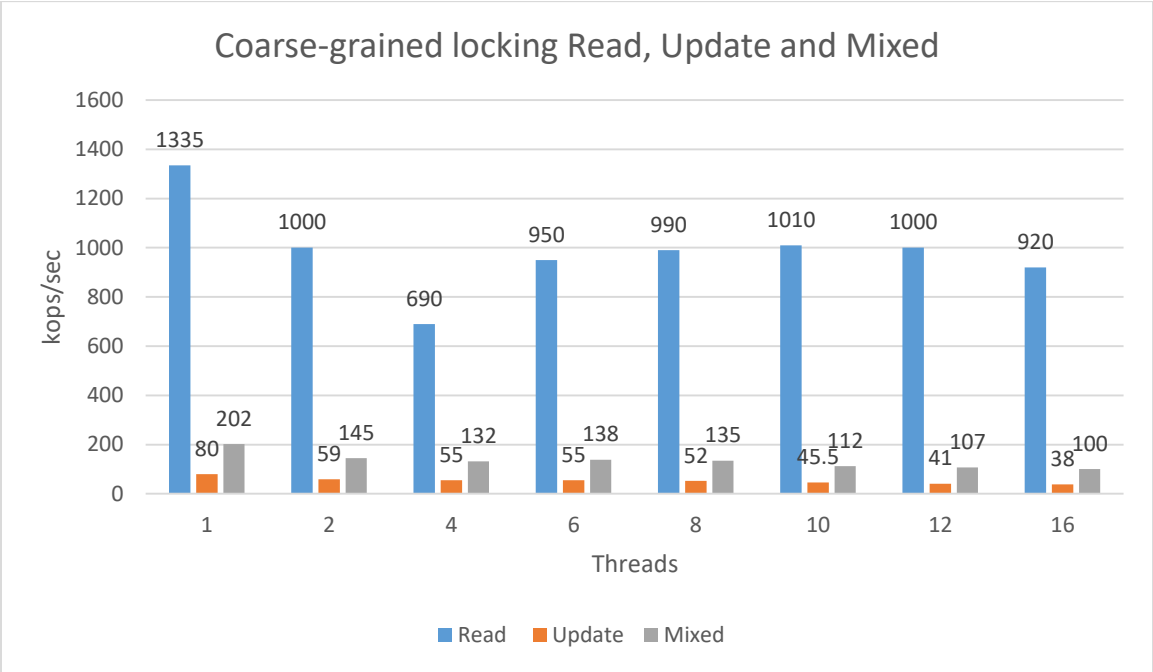
Below, we have highlighted the code that corresponds to that kind of unexpected behavior. It is apparent, that by not locking the remove accesses, one node gets deleted by one thread, while another thread requests to read it but doesn't exist anymore.

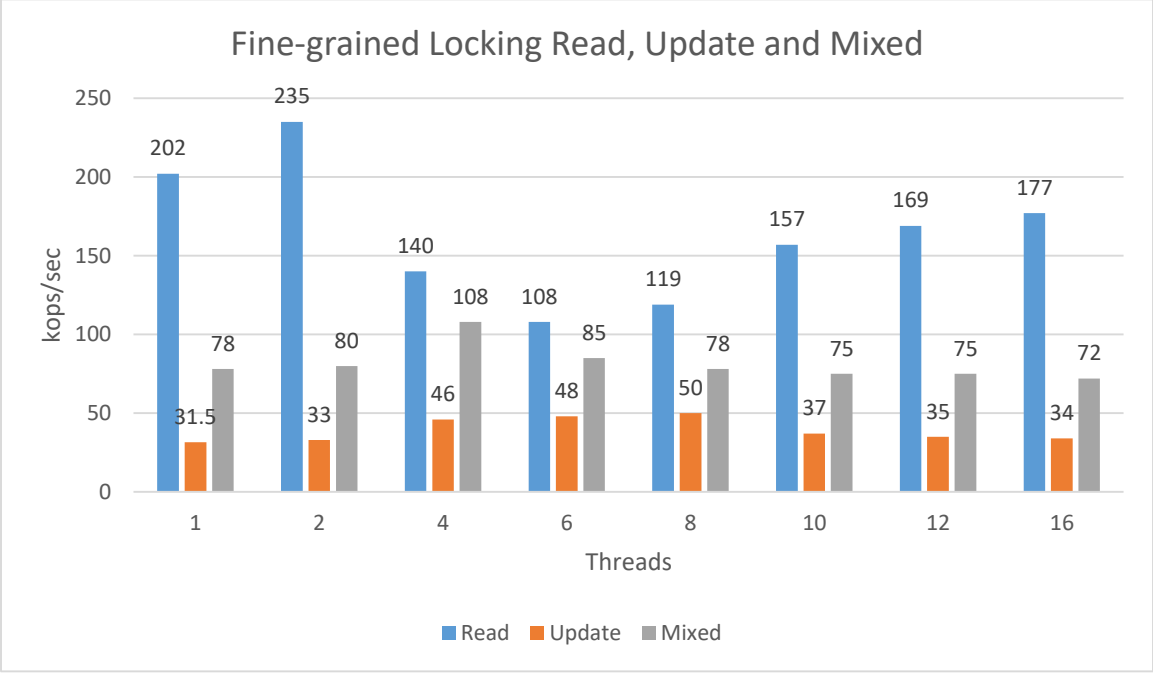
```
/* insert v into the list */
void insert(T v) {
   ins_lock.lock();
   node<T>* pred = nullptr;
   node<T>* succ = first;
   //things to do while inserting
   ins_lock.unlock();
}
```

```
/* remove v from the list */
void remove(T v) {
   //ins_lock.lock();
   node<T>* pred = nullptr;
   node<T>* succ = first;
   //things to do while removing
   if(!found){
      //ins_lock.unlock;
      return;
   }
   delete current;
   //ins_lock.unlock();
}
```

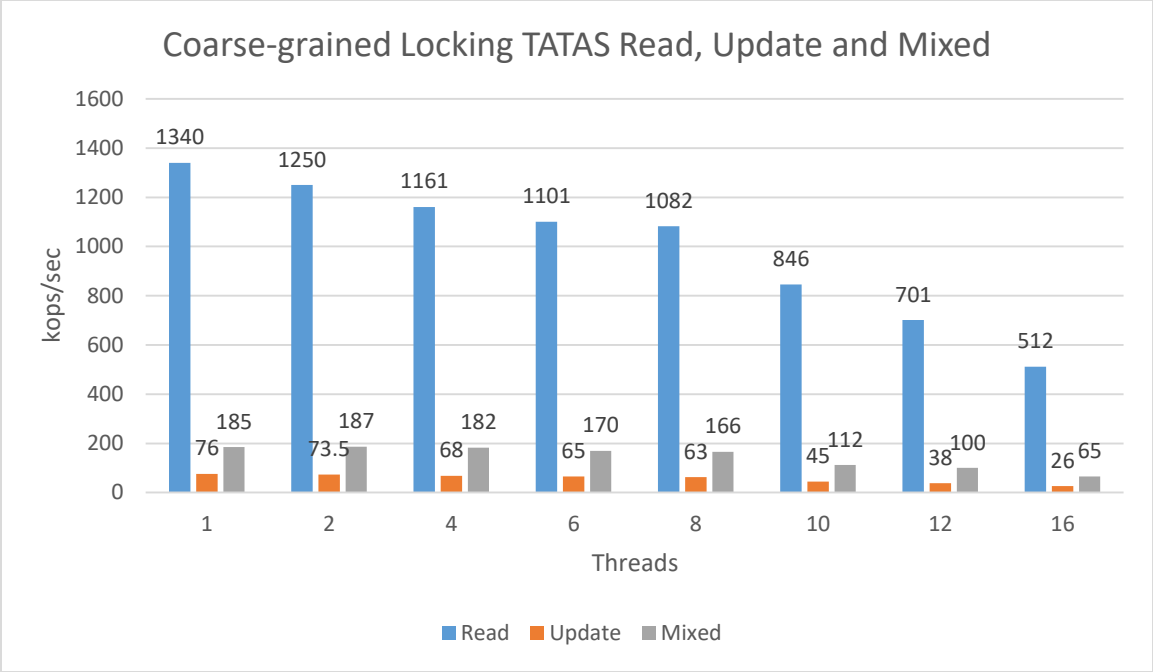

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

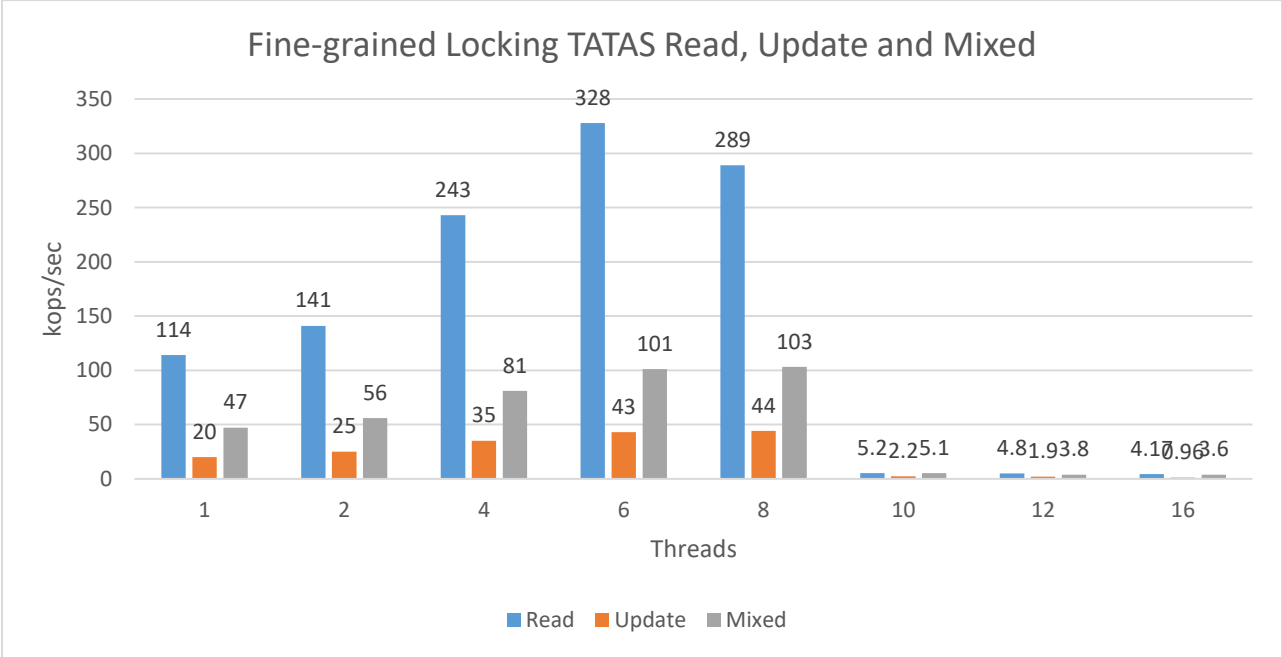
Result figures



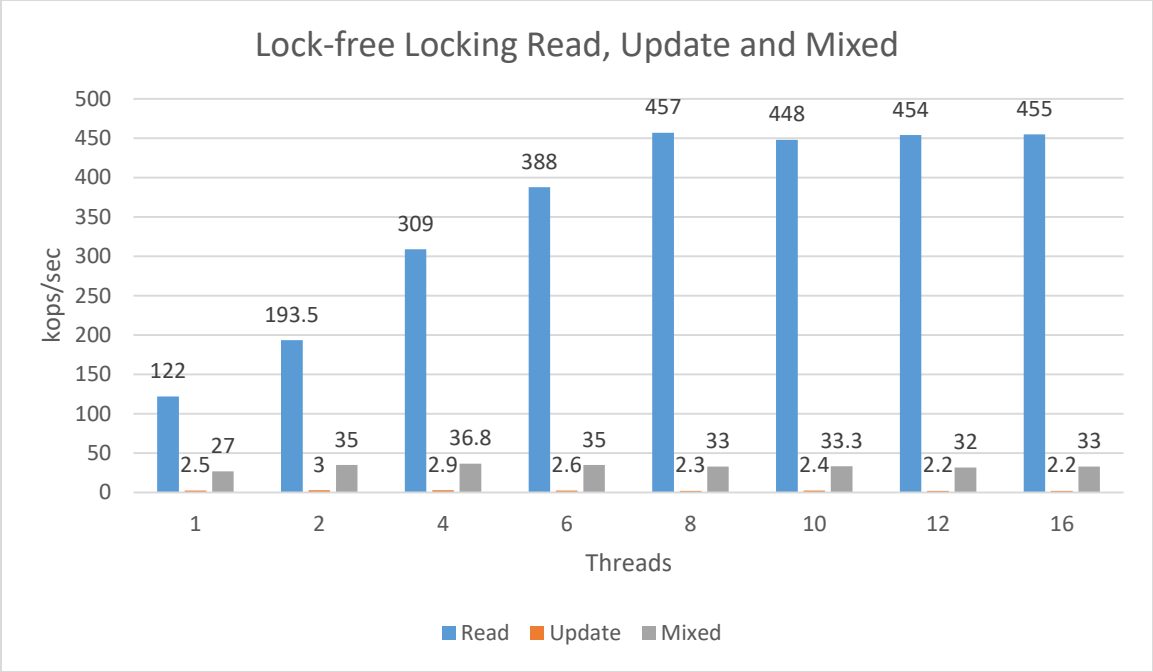


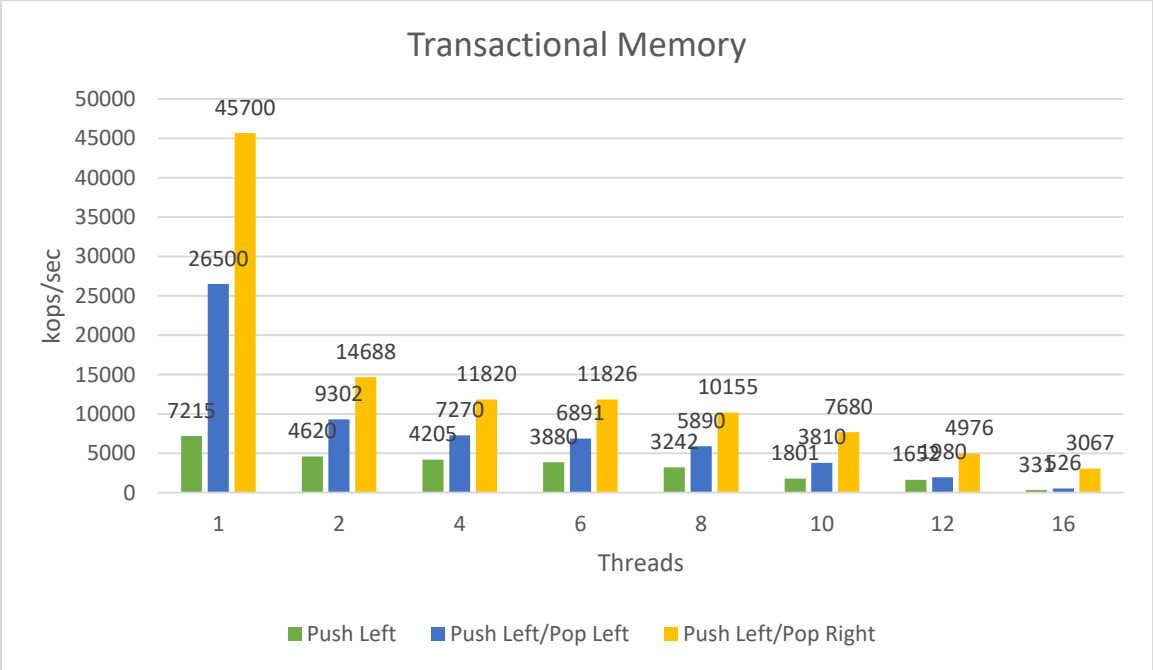
Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7





Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7





Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

Performance evaluation

Basic axes of evaluating benchmarking

Coarse vs. fine

Comparing the two charts of Coarse and Fine-grained locking we observe a main difference and a main similarity. The basic difference is the amount of operations that are running every second. In the Coarse-grained locking, we achieved about 1 million operations per second in the "read" function of the benchmark, while in the Fine grained technique this number falls to about 150.000. The numbers achieved in the "update" and "mixed" functions, are about the same yielding about 135.000 in update and 50.000 in mixed. The similarity lies in the fact that as the number of threads is increasing, there is a drop in the number of operations that we achieve every second. This happens for 2 and 4 threads in Coarse-grained and then we have a small steady increase as the threads are increasing. The same behavior is observed in Fine-grained when we use 2, 4 and 6 threads.

Mutex vs. spinlock

Now, comparing the previous mutex implementation with our own spinlock. We have managed to implement a spinlock that performs just a bit better than a mutex lock in a single-threaded execution. From 1.335.000 operations per second in Coarse-grained we reached 1.340.000 operations per second for the read function. Besides that, in the coarse-grained scheme the TATAS spinlock seems more stable. From 1.340.000 operations per second in the single threaded execution we achieved 1.082.000 operations per second using 8 threads, always yielding more operations per second than the mutex implementation in the same number of threads (e.g. 8 thread mutex execution gives as 990.000 operations per second). As far as the fine-grained concept, our spinlock performs worse than a simple mutex lock in 1 or 2 threaded execution, it scales remarkably for more threads.

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

Mutex vs. spinlock vs. lock-free

Introducing the Lock-free technique in comparison with the mutex locking and the spinlock that we implemented, we can see that it is by far the best of them all as far as the scaling is concerned. We observe that as the number of threads increases the number of operations executed is growing constantly reaching about 457.000 in 8 threads for the “read” function. Something totally reasonable, since our Read Copy Update technique is reader-friendly. Nevertheless, we should not disregard the fact that the number of operations executed in the other two functions (“update” and “mixed”) is the lowest of them all. We barely achieved 3.000 operations/sec in update and 35.000 operations/sec in mixed. Respectively, this is something expected because of the continuous copying of the structure.

Mutex vs. spinlock vs. lock-free vs. transactional

In the final part we have the results of a queue running in contrast with the previous experiments where we had a list, either simply or doubly linked. Nevertheless, the mentality is quite different mostly in terms of difficulty and complexity of the codes. In this case we approach concurrency through the use of transactional memory and the results are quite fascinating. Although we still have a significant drop in the number of operations executed as the number of threads is increasing however the difference in terms of performance is big enough compared to the previous techniques despite their native algorithmic complexity differences. Characteristically we achieved a maximum of 45.700.000 operations per second in the PushLeft/PopRight function, 26.500.000 operations per second in the PushLeft/PopLeft function and 7.215.000 operations per second in the PushLeft function using one thread. As we increase the number of threads, the number of operations that are executed is falling considerably, but this is not enough to tarnish the abilities of this technique. As we can see, even at the lowest level at 8 threads we have about 4.976.000 operations per second executed in the PushLeft/PopRight function, a number that is by far better than the best estimates of all the other techniques used. A way to reason about this is that operations are being executed on different ends of the queue, thus enabling better circulation of readers or writers.

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

Overall Comparison

As we can observe from the numbers, coarse grain lock works better with only one thread working, and that seems accurate as the list gets locked for only one thread every time, but as we increase the number of threads we have a slowdown. On the other side, on the fine grain locking we can observe an actual speedup pretty serious in the most cases, proving that this concept actually works as the number of thread increases.

Using tatas lock with coarse grain, we get slightly better results comparatively to the coarse grain locking, but overall as number of thread increases we don't get a slowdown as the concept of coarse graining imposes. Fine grain with tatas lock works poorly with low number of threads, but as we increase them we actually see some nice speedup.

As for the lock-free list it works perfectly for concurrent readers, as it allows all of them to read the list concurrently not minding particularly of this action, though it works poorly when inserts and deletes are happening as we stated above. There have been some solutions about this problem in the lock free list in which first of all copying the whole list is not needed. Besides the rest, Harris has come up with an algorithm for a different implementation such as: place a 'mark' in the next pointer of the soon-to-be deleted node, fail when we try to CAS the 'mark', when detected go back to start of the list and restart. Of course, we can proceed with plenty of implementations.

Transactional Memory performance

In all the implementations, we see a negative speed-up compared to running the program on one thread. We think this happens because the queue implementation works by adding or removing elements from/to the edges, and this causes a lot of contention and transactions to fail when there is more than one thread. When only PushLeft() function is called, the program executes slowest compared to the others. When PushLeft() and PopLeft() functions are subsequently called from each thread, more operations are possible.

We benchmarked PushLeft() and PopLeft() functions separately for an explanation, and we have noticed that the popping operation is much faster compared to the pushing operation. This probably happens because we are

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

allocating new memory dynamically when pushing new elements. The third case (P3) is faster at all points compared to the second case (P2). This is expected and easily explained by how the transactions work. In this case the push and the pop functions are operating on the different parts of the queue, and their transactions do not affect each other.

Ease of implementation / Reasoning for performance issues

COARSE

On the advantages: it was easy to perceive and implement; it is faster and easier to implement operations that access multiple locations because they are all guarded by the same lock, easier to implement modifications on the data structure shape and obviously, the concept seems correct while on The disadvantages: the list isn't really used concurrently as threads "stand in line" to use the list, leading to unnecessary blocking, and creating a sequential bottleneck, plus adding more threads doesn't seem to improve throughput. Finally, works poorly with contention.

FINE

On the advantages: manages to have more concurrent accesses on the list than the coarse grain lock of course, because threads can traverse in parallel thus improves performance, while on

The disadvantages it was obviously harder to implement – spend a respectable amount of time detecting and fixing data races- we get a long chain of acquiring and releasing lock, making it no so efficient after all. Takes up more space

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

TATAS COARSE and FINE GRAINED

- Spinlock implementation

Implementing a spinlock is not an easy job. On the contrary, one must be really careful, study also some basic architecture stuff, and be aware of the quality of the result. After studying thoroughly about caching effects and the implementation of corresponding memory library of gcc we managed to have the desired effects. Bottom line, it is difficult to implement even a descent spinlock, but the results of success are rewarding.

- Integration on the above techniques

In order to integrate the TATAS technique, the only thing that we had to do, was to change our first coarse and fine grain lock implementation of the list. That means that we had to replace our mutexes used, with our tatas_lock. The rest remained the same in terms of logic behind locking the concurrent data structure.

LOCK FREE

On the advantages: although it may seem difficult to implement, it was easier than the fine-grained locking, although some parts were tricky, not using locks and not locking unlocking and/or waiting for a lock seems to be faster at first sight.

On the disadvantages: by no means is it efficient to copy an entire list at least one time for inserting or deleting a node, making this mechanism friendly for concurrent readers only.

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

TRANSACTIONAL MEMORY

This paradigm corresponds to actual hardware. Despite that, the library provided by gcc literally works as a language abstraction. It is created in such a way that the programmer should not change his course of thought while implementing, and also that he should not be bothered with many details regarding how to transform his data structure (use responsibly! Only when suitable of course!) to one with capability of handling concurrent accesses. So naturally enough, the course of implementing and handling of our data structure did not differ at all from a sequential one. Only difference is that we wrapped around our critical parts an atomic transaction and all of our blocks' code was automatically considered to happen atomically, and even better with ACID properties as discussed above!

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

References

1. Barney, B. (2010). Introduction to parallel computing. Lawrence Livermore National Laboratory.
2. Hennessy, J. L., & Patterson, D. A. (2011). Computer architecture: a quantitative approach. Elsevier.
3. Almasi, G. S., & Gottlieb, A. (1988). Highly parallel computing.
4. Adve, S., Adve, V. S., Agha, G., Frank, M. I., Garzarán, M., Hart, J. & Marinov, D. (2008). Parallel computing research at Illinois: The UPCRC agenda. Urbana, IL: Univ. Illinois Urbana-Champaign.
5. Singh, A., & Singh, S. P. (2013). Terminology and taxonomy parallel computing architecture. ASIAN JOURNAL OF COMPUTER SCIENCE & INFORMATION TECHNOLOGY, 1(5).
6. Asanovic, K., Bodik, R., Catanzaro, B. C., Gebis, J. J., Husbands, P., Keutzer, K. & Yelick, K. A. (2006). The landscape of parallel computing research: A view from berkeley (Vol. 2). Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley.
7. Pike, R., & Gerrand, A. (2012). Concurrency is not parallelism. Heroku Waza.
8. David, A. P., & John, L. H. (2005). Computer organization and design: the hardware/software interface. San mateo, CA: M organ Kaufmann Publishers, 1, 998.
9. *Operating System Concepts* 9th edition, Abraham Silberschatz. "Chapter 4: Threads"
10. Schneider, F. B. (2012). On concurrent programming. Springer Science & Business Media.
11. http://www.emu.edu.tr/aelci/courses/d-318/d-318-files/plbook/concurre.htm
12. Shavit, N., & Taubenfeld, G. (2016). The computability of relaxed data structures: queues and stacks as examples. Distributed Computing
13. Herlihy, M. P., & Wing, J. M. (1990). Linearizability: A correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems (TOPLAS)
14. www.wikipedia.org

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

15. Kukanov, Alexey (2008-03-04). "Why a simple test can get parallel slowdown"
16. https://cs.stackexchange.com/questions/55433/what-is-meant-by-superlinear-speedup-is-it-possible-to-have-superlinear-speedup
17. Grama, A. (2003). Introduction to parallel computing. Pearson Education.
18. Xavier, C., & Iyengar, S. S. (1998). Introduction to parallel algorithms (Vol. 1). John Wiley & Sons.
19. Loebner, N. (2006). Senior Project: Parallel Programming.
20. Goodacre, J. & Sloss, A.N. 2005, "Parallelism and the ARM instruction set architecture", Computer, vol. 38, no. 7, pp. 42-50.
21. Culler, D. E., Singh, J. P., & Gupta, A. (1999). Parallel computer architecture: a hardware/software approach. Gulf Professional Publishing.
22. Harris, T., Larus, J., & Rajwar, R. (2010). Transactional memory (synthesis lectures on computer architecture). Synthesis Lectures on Computer Architecture. Morgan and Claypool.
23. "Transactional Memory: History and Development". *Kukuruku Hub*. Retrieved
24. Parri, J. (2010). An introduction to transactional memory. ELG7187 Topics In Computers: Multiprocessor Systems On Chip, fall.
25. Hwang, K., & Jotwani, N. (2011). Advanced Computer Architecture, 3e. McGraw-Hill Education.
26. Peter Bright. AMD's "heterogeneous Uniform Memory Access" coming this year in Kaveri, Ars Technica.
27. Blagodurov, S., Zhuravlev, S., Fedorova, A., & Kamali, A. (2010, September). A case for NUMA-aware contention management on multicore systems. In Proceedings of the 19th international conference on Parallel architectures and compilation techniques. ACM.
28. Majo, Z., & Gross, T. R. (2011, May). Memory system performance in a NUMA multicore multiprocessor. In Proceedings of the 4th Annual International Conference on Systems and Storage. ACM.
29. *"Intel Dual-Channel DDR Memory Architecture White Paper" (PDF) (Rev. 1.0 ed.). Infineon Technologies North America and Kingston Technology.*
30. Lamport, L. (1979). How to make a multiprocessor computer that correctly executes multiprocess progranm. IEEE transactions on computers.
31. Saltzer, J. H. (1966). Traffic control in a multiplexed computer system (Doctoral dissertation, Massachusetts Institute of Technology).
32. Single-Threading: Back to the Future? Sergey Ignatchenko, Overload

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

33. Shavit, N., & Moir, M. (2007). Concurrent data structures. Handbook of Data Structures and Applications.
34. Dijkstra, E. W. (1965). "Solution of a problem in concurrent programming control". *Communications of the ACM*.
35. *"PODC Influential Paper Award: 2002", ACM Symposium on Principles of Distributed Computing*
36. https://timharris.uk/papers/2001-disc.pdf
37. Holzmann, G. J., & Bosnacki, D. (2007). The design of a multicore extension of the SPIN model checker.
38. Silberschatz, A. (1994). Peter B. galvin. Operating system concepts, 4th ed., reading, MA: Addision-wesley.
39. Guniguntala, D., McKenney, P. E., Triplett, J., & Walpole, J. (2008). The read-copy-update mechanism for supporting real-time applications on shared-memory multiprocessor systems with Linux. IBM Systems Journal.
40. www.stackoverflow.com
41. Artho, C., Havelund, K., & Biere, A. (2003). High-level data races. Software Testing, Verification and Reliability.
42. Coulouris, G. F., Dollimore, J., & Kindberg, T. (2005). Distributed systems: concepts and design. pearson education.
43. Padua, D. (Ed.). (2011). Encyclopedia of parallel computing. Springer Science & Business Media.
44. Silberschatz, A., Galvin, P. B., & Gagne, G. (2006). Operating system principles. John Wiley & Sons.
45. Schneider, G. M., & Gersting, J. (2018). Invitation to computer science. Cengage Learning.
46. Tanenbaum, A. S. (1995). Distributed operating systems. Pearson Education India.
47. Đorđević-Kajan, S. (2001). Tanenbaum Andrew S.: Modern operating systems , Prentice-Hall, Upper Saddle River, New Jersey, USA, 2001. Facta universitatis-series: Electronics and Energetics
48. *Herlihy, Maurice; Shavit, Nir (2012). The Art of Multiprocessor Programming. Elsevier.*
49. Raynal, M. (2012). Concurrent programming: algorithms, principles, and foundations. Springer Science & Business Media.
50. Reinders, James (10 September 2007). "Understanding task and data parallelism | ZDNet"

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7

51. Quinn, Michael J. (2007). Parallel programming in C with MPI and openMP (Tata McGraw-Hill ed.). New Delhi: Tata McGraw-Hill Pub
52. Hicks, Michael. "Concurrency Basics"
53. Nakul Manchanda; Karan Anand (2010-05-04). "Non-Uniform Memory Access (NUMA)" (PDF). New York University.
54. Ami Marowka, “A Study of the Usability of Multicore Threading Tools”, International Journal of Software Engineering and Its Applications, Vol. 4, No.3, 2010
55. Leslie Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs", IEEE Trans. Comput. C-28,9 (Sept. 1979), 690-691.
56. Sarita V. Adve, Kourosh Gharachorloo, "Shared Memory Consistency Models: A Tutorial"

Institutional Repository - Library & Information Centre - University of Thessaly
12/07/2018 22:00:19 EEST - 137.108.70.7