Parallel Computing Concepts

CSInParallel Project

July 26, 2012

CONTENTS

CHAPTER

ONE

INTRODUCTION

1.1 Motivation

Moore's "Law": an empirical observation by Intel co-founder Gordon Moore in 1965. The number of components in computer circuits had doubled each year since 1958, and Moore predicted that this doubling trend would continue for another decade. Incredibly, over four decades later, that number has continued to double each two years or less.

However, since about 2005, it has been impossible to achieve such performance improvements by making larger and faster single CPU circuits. Instead, the industry has created *multi-core* CPUs – single chips that contain multiple circuits for carrying out instructions (cores) per chip.

The number of cores per CPU chip is growing exponentially, in order to maintain the exponential growth curve of Moore's Law. But most **software** has been designed for single cores.

Therefore, CS students must learn principles of parallel computing to be prepared for careers that will require increasing understanding of how to take advantage of multi-core systems.

1.2 Some pairs of terms

parallelism multiple (computer) actions physically taking place at the same time

concurrency programming in order to take advantage of parallelism (or virtual parallelism)

> **Comments** Thus, parallelism takes place in hardware, whereas concurrency takes place in software. Operating systems must use concurrency, since they must manage multiple processes that are abstractly executing at the same time–and can physically execute at the same time, given parallel hardware (and a capable OS).

process the execution of a program

thread a sequence of execution within a program

> **Comments** Every process has at least one thread of execution, defined by that process's program counter. If there are multiple threads within a process, they share resources such as the process's memory allocation. This reduces the computational overhead for switching among threads (also called *lightweight processes*), and enables efficient sharing of resources (e.g., communication through shared memory locations).

sequential programming programming for a single core

concurrent programming programming for multiple cores or multiple computers

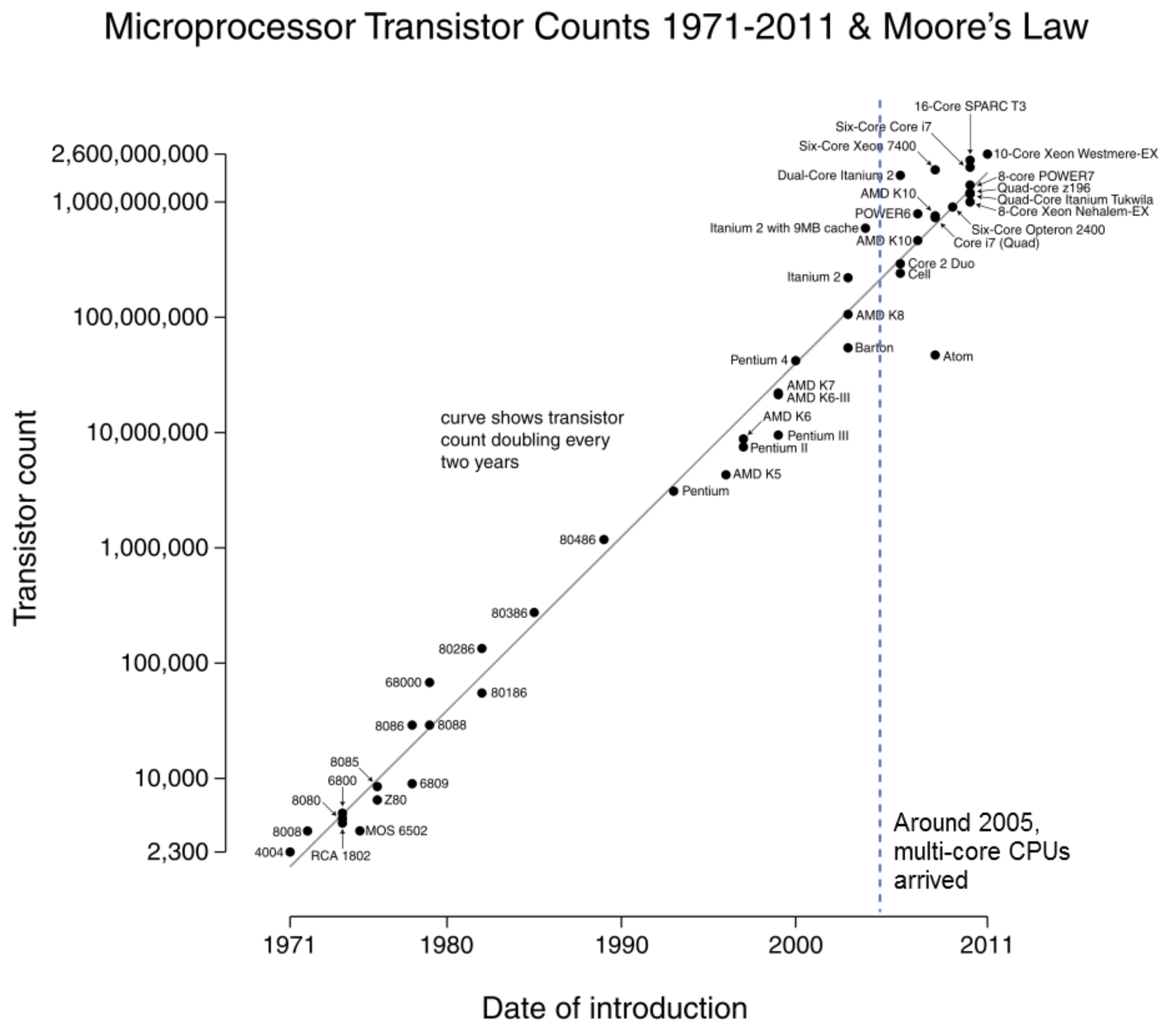

Figure 1.1: Plot of CPU transistor counts against dates of introduction. Note the logarithmic vertical scale; the line corresponds to exponential growth with transistor count doubling every two years. This figure is from Wikimedia Commons.

Comments CS students have primarily learned sequential programming in the past. These skills are still relevant, because concurrent programs ordinarily consist of sets of sequential programs intended for various cores or computers.

multi-core computing computing with systems that provide multiple computational circuits per CPU package

distributed computing computing with systems consisting of multiple computers connected by computer network(s)

Comments Both of these types of computing may be present in the same system (as in our MistRider and Helios clusters).

data parallelism the same processing is applied to multiple subsets of a large data set in parallel

task parallelism different tasks or stages of a computation are performed in parallel

Comments A telephone call center illustrates data parallelism: each incoming customer call (or outgoing telemarketer call) represents the services processing on different data. An assembly line (or computational pipeline) illustrates task parallelism: each stage is carried out by a different person (or processor), and all persons are working in parallel (but on different stages of different entities.)

shared memory multiprocessing e.g., multi-core system, and/or multiple CPU packages in a single computer, all sharing the same main memory

cluster multiple networked computers managed as a single resource and designed for working as a unit on large computational problems

grid computing distributed systems at multiple locations, typically with separate management, coordinated for working on large-scale problems

cloud computing computing services are accessed via networking on large, centrally managed clusters at data centers, typically at unknown remote locations

SETI@home another example of distributed computing

Comments Although multi-core processors are driving the movement to introduce more parallelism in CS courses, distributed computing concepts also merit study. For example, Intel's recently announced 48-core chip for research behaves like a distributed system with regards to interactions between its cache memories.

CHAPTER

TWO

PARALLEL SPEEDUP

2.1 Introduction

The *speedup* of a parallel algorithm *over* a corresponding sequential algorithm is the ratio of the compute time for the sequential algorithm to the time for the parallel algorithm. If the speedup factor is *n*, then we say we have *n-fold* speedup. For example, if a sequential algorithm requires 10 min of compute time and a corresponding parallel algorithm requires 2 min, we say that there is 5-fold speedup.

The observed speedup depends on all implementation factors. For example, more processors often leads to more speedup; also, if other programs are running on the processors at the same time as a program implementing a parallel algorithm, those other programs may reduce the speedup. Even if a problem is embarrassingly parallel, one seldom actually obtains n-fold speedup when using n-fold processors. There are a couple of explanations for this occurrence:

- There is often *overhead* involved in a computation. For example, in the solar system computation, results need to be copied across the network upon every iteration. This communication is essential to the algorithm, yet the time spend on this communication does not directly compute more solutions to the n-body problem. In general, communication costs are frequent contributors to overhead. The processing time to schedule and dispatch processes also leads to overhead.
- *trues* occur when a process must wait for another process to deliver computing resources. For example, after each computer in the solar system computation delivers the results of its iteration, it must wait to receive the updated values for other planets before beginning its next iteration.
- Some parts of a computation may be inherently sequential. In the polar ice example, only the matrix computation was parallelized, and other parts gained in performance only because they were performed on faster hardware and software (on a single computer)

On rare occasions, using *n* processors may lead to *more* than an *n*-fold speedup. For example, if a computation involves a large data set that does not fit into the main memory of a single computer, but *does* fit into the collective main memories of *n* computers, and if an embarrassingly parallel implementation requires only proportional portions of the data, then the parallel computation involving *n* computers may run more than *n* times as fast because disk accesses can be replaced by main-memory accesses.

Replacing main-memory accesses by cache accesses could have a similar effect. Also, parallel pruning in a backtracking algorithm could make it possible for one process to avoid an unnecessary computation because of the prior work of another process.

2.2 Amdahl's Law

Amdahl's Law is a formula for estimating the maximum speedup from an algorithm that is part sequential and part parallel. The search for *2k*-digit primes illustrates this kind of problem: First, we create a list of all *k*-digit primes, using a sequential sieve strategy; then we check *2k*-digit random numbers in parallel until we find a prime.

The Amdahl's Law formula is

$$\textit{overall speedup} = \frac{1}{(1-P)+\frac{P}{S}}$$

- P is the time proportion of the algorithm that can be parallelized.
- S is the speedup factor for that portion of the algorithm due to parallelization.

For example, suppose that we use our strategy to search for primes using 4 processors, and that 90% of the running time is spent checking 2k-digit random numbers for primality (after an initial 10% of the running time computing a list of k-digit primes). Then P = .90 and S = 4 (for 4-fold speedup). According to Amdahl's Law,

$$\textit{overall speedup} = \frac{1}{(1-0.90)+\frac{0.90}{4}} = \frac{0.10}{0.225} = 3.077$$

This estimates that we will obtain about 3-fold speedup by using 4-fold parallelism.

Note:

- Amdahl's Law computes the overall speedup, taking into account that the *sequential* portion of the algorithm has no speedup, but the *parallel* portion of the algorithm has speedup *S*.
- It may seem surprising that we obtain only 3-fold overall speedup when 90% of the algorithm achieves 4-fold speedup. This is a lesson of Amdahl's Law: the *non-parallelizable* portion of the algorithm has a disproportionate effect on the overall speedup.
- A non-computational example may help explain this effect. Suppose that a team of four students is producing a report, together with an executive summary, where the main body of the report requires 8 hours to write, and the executive summary requires one hour to write and must have a single author (representing a sequential task). If only one person wrote the entire report, it would require 9 hours. But if the four students each write 1/4 of the body of the report (2 hours, in 4-fold parallelism), then one student writes the summary, then the elapsed time would be 3 hours—for a 3-fold overall speedup. The sequential portion of the task has a disproportionate effect because the other three students have nothing to do during that portion of the task.

A short computation shows why Amdahl's Law is true.

- Let T_s be the compute time without parallelism, and T_p the compute time *with* parallelism. Then, the speedup due to parallelism is

  $$\textit{total speedup} = \frac{T_s}{T_p}$$

- The value P in Amdahl's Law is the proportion of T_s that can be parallelized, a number between 0 and 1. Then, the proportion of T_s that cannot be parallelized is 1-P.
- This means that

  $$T_p = \textit{time spent in unparallelizable code} + \textit{time spent in parallelizable code} = (1-P)\times T_s + P \times \frac{T_s}{S}$$

- We conclude that

  $$\textit{total speedup} = \frac{T_s}{T_p} = \frac{T_s}{(1-P)\times T_s + P\times\frac{T_s}{S}} = \frac{1}{(1-P)+\frac{P}{S}}$$

CHAPTER

THREE

SOME OPTIONS FOR COMMUNICATION

In simple data parallelism, it may not be necessary for parallel computations to share data with each other during the executions of their programs. However, most other forms of concurrency require communication between parallel computations. Here are three options for communicating between various processes/threads running in parallel.

1. **message passing** - communicating with basic operations **send** and **receive** to transmit information from one computation to another.
2. **shared memory** - communicating by reading and writing from local memory locations that are accessible by multiple computations
3. **distributed memory** - some parallel computing systems provide a service for sharing memory locations on a remote computer system, enabling non-local reads and writes to a memory location for communication.

   Comments For distributed systems, message passing and distributed memory (if available) may be used. All three approaches may be used in concurrent programming in multi-core parallelism. However, shared (local) memory access typically offers an attractive speed advantage over message passing and remote distributed memory access.

When multiple processes or threads have both read and write access to a memory location, there is potential for a **race condition**, in which the correct behavior of the system depends on timing. (Example: filling a shared array, with algorithms for accessing and updating a variable *nextindex*.)

- **resource** - a hardware or software entity that can be allocated to a process by an operating system

  Comments A memory location is an example of a (OS) resource; other examples are: files; open files; network connections; disks; GPUs; print queue entries. Race conditions may occur around the handling of any OS resource, not just memory locations.

- In the study of Operating Systems, **inter-process communication (IPC)** strategies are used to avoid problems such as race conditions. Message passing is an example of an IPC strategy. (Example: solving the array access problem with message passing.)
- Message passing can be used to solve IPC problems in a distributed system. Other common IPC strategies (semaphores, monitors, etc.) are designed for a single (possibly multi-core) computer.

CHAPTER

FOUR

SOME ISSUES IN CONCURRENCY

We will use the Hadoop implementation of map-reduce for clusters as a running example.

Fault tolerance is the capacity of a computing system to continue to satisfy its spec in the presence of faults (causes of error)

Comments With more parallel computing components and more interactions between them, more faults become possible. Also, in large computations, the cost of restarting a computation may become greater. Thus, fault tolerance becomes more important and more challenging as one increases the use of parallelism. Systems (such as map-reduce) that automatically provide for fault tolerance help programmers of parallel systems become more productive.

Mutually exclusive access to shared resources means that at most one computation (process or thread) can access a resource (such as a shared memory location) at a time. This is one of the requirements for correct IPC. One approach to mutually exclusive access is locking, in which a mechanism is provided for one computation to acquire a "lock" that only one computation may hold at any given time. A computation possessing the lock may then use that lock's resource without fear of interference by another process, then release the lock when done.

Comments Designing computationally correct locking systems and using them correctly for IPC can often be quite tricky.

Scheduling means assigning computations (processes or threads) to processors (cores, distributed computers, etc.) according to time. For example, in map-reduce computing, we mapper processes are scheduled to particular cluster nodes having the necessary local data; they are rescheduled in the case of faults; reducers are scheduled at a later time.

INDEX