SCS5108 Parallel Systems Unit I

The changing roles of parallelism

Parallel computing is a type of computation in which many calculations are carried out simultaneously, operating on the principle that large problems can often be divided into smaller ones, which are then solved at the same time. There are several different forms of parallel computing: bit-level, instruction-level, data, and task parallelism. Parallelism has been employed for many years, mainly in high-performance computing, but interest in it has grown lately due to the physical constraints preventing frequency scaling. As power consumption (and consequently heat generation) by computers has become a concern in recent years, parallel computing has become the dominant paradigm in computer architecture, mainly in the form of multi-core processors.

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). 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 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, 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.

Metrics:

In computer architecture, **speedup** is a metric for improvement in performance between two systems processing the same problem. More technically, is it 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.

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

- v 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 frequent unit of throughput is the instruction per cycle (IPC). Its reciprocal, the cycle per instruction (CPI), is another frequent unit of latency.

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

Speedup in latency

Speedup in *latency* is defined by the following formula:

$$S_{\mathbf{latency}} = \frac{L_1}{L_2} = \frac{T_1 W_2}{T_2 W_1},$$

where

- S_{latency} is the speedup in latency of the architecture 2 with respect to the architecture 1;

- L_1 is the latency of the architecture 1;
- L_2 is the latency of the architecture 2.

Speedup in latency can be predicted from Amdahl's law or Gustafson's law.

Speedup in throughput

Speedup in *throughput* is defined by the following formula:

$$S_{\mathbf{throughput}} = \frac{Q_2}{Q_1} = \frac{\rho_2 A_2 T_1 W_2}{\rho_1 A_1 T_2 W_1} = \frac{\rho_2 A_2}{\rho_1 A_1} S_{\mathbf{latency}},$$

where

- $S_{\text{throughput}}$ is the speedup in throughput of the architecture 2 with respect to the architecture 1;
- Q_1 is the throughput of the architecture 1;
- Q_2 is the throughput of the architecture 2.

Amdahl's law

In computer architecture, **Amdahl's law** (or **Amdahl's argument**) 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 can be formulated the following way:

$$S_{\mathbf{latency}}(s) = \frac{1}{1 - p + \frac{p}{s}},$$

where

- S_{latency} is the theoretical speedup in latency of the execution of the whole task;
- s is the speedup in latency of the execution of the part of the task that benefits from the improvement of the resources of the system;
- p is the percentage of the execution time of the whole task concerning the part that benefits from the improvement of the resources of the system *before the improvement*.

Furthermore,

$$\begin{cases} S_{\mathbf{latency}}(s) \leq \frac{1}{1-p} \\ \lim_{s \to \infty} S_{\mathbf{latency}}(s) = \frac{1}{1-p}. \end{cases}$$

show that the theoretical speedup of the execution of the whole task increases with the improvement of the resources of the system and that regardless the magnitude of the improvement, the theoretical speedup is always limited by the part of the task that cannot benefit from the improvement.

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 is relevant only for a low number of processors and very parallelizable programs.

Gustafson's law

In computer architecture, **Gustafson's law** (or **Gustafson–Barsis's law**) gives the theoretical speedup in latency of the execution of a task *at fixed execution time* that can be expected of a system whose resources are improved. It is named after computer scientist John L. Gustafson and his colleague Edwin H. Barsis, and was presented in the article *Reevaluating Amdahl's Law* in 1988.

Gustafson's law can be formulated the following way:

$$S_{\text{latency}}(s) = 1 - p + sp,$$

where

- S_{latency} is the theoretical speedup in latency of the execution of the whole task;
- s is the speedup in latency of the execution of the part of the task that benefits from the improvement of the resources of the system;
- p is the percentage of the execution workload of the whole task concerning the part that benefits from the improvement of the resources of the system *before the improvement*.

Gustafson's law addresses the shortcomings of Amdahl's law, which is based on the assumption of a fixed problem size, that is of an execution workload that does not change with respect to the improvement of the resources. Gustafson's law instead proposes that programmers tend to set the size of problems to fully exploit the computing power that becomes available as the resources improve. Therefore, if faster equipment is available, larger problems can be solved within the same time.

The impact of Gustafson's law was to shift research goals to select or reformulate problems so that solving a larger problem in the same amount of time would be possible. In a way the law redefines efficiency, due to the possibility that limitations imposed by the sequential part of a program may be countered by increasing the total amount of computation.

Scalability :

Scalability is the capability of a system, network, or process to handle a growing amount of work, or its potential to be enlarged in order to accommodate that growth. For example, it can refer to the capability of a system to increase its total output under an increased load when resources (typically hardware) are added. An analogous meaning is implied when the word is used in an economic context, where scalability of a company implies that the underlying business model offers the potential for economic growth within the company.

Scalability, as a property of systems, is generally difficult to define and in any particular case it is necessary to define the specific requirements for scalability on those dimensions that are deemed important. It is a highly significant issue in electronics systems, databases, routers, and networking. A system whose performance improves after adding hardware, proportionally to the capacity added, is said to be a ***scalable system***.

An algorithm, design, networking protocol, program, or other system is said to *scale* if it is suitably efficient and practical when applied to large situations (e.g. a large input data set, a large number of outputs or users, or a large number of participating nodes in the case of a distributed system). If the design or system fails when a quantity increases, it *does not scale*. In practice, if there are a large number of things (n) that affect scaling, then resource requirements (for example, algorithmic time-complexity) must grow less than n^2 as n increases. An example is a search engine, which scales not only for the number of users, but also for the number of objects it indexes. Scalability refers to the ability of a site to increase in size as demand warrants.

The concept of scalability is desirable in technology as well as business settings. The base concept is consistent – the ability for a business or technology to accept increased volume without impacting the contribution margin (= revenue – variable costs). For example, a given piece of equipment may have a capacity for 1–1000 users, while beyond 1000 users additional equipment is needed or performance will decline (variable costs will increase and reduce contribution margin).

Scalable Computing Towards Massive Parallelism

- *Levels of Parallelism*
  - *Bit-level parallelism (BLP)*
  - *instruction-level parallelism (ILP)*
  - *Data-level parallelism (DLP)*
  - *task-level parallelism (TLP)*
  - *job-level parallelism (JLP)*
- *Key issues of the age of Internet computing*

- *Efficiency measured in building blocks and execution model to exploit massive parallelism as in HPC. This may include data access and storage model for HTC and energy efficiency.*
- *Dependability in terms of reliability and self-management from the chip to system and application levels. The purpose is to provide high-throughput service with QoS assurance even under failure conditions.*
- *Adaptation in programming model which can support billions of job requests over massive datasets, virtualized cloud resources, and flexible application service model.*

Parallel Architecture

A parallel computer is a collection of processing elements that cooperate to solve large problems fast
Some broad issues:

• Resource Allocation:
– how large a collection?
– how powerful are the elements?
– how much memory?
• Data access, Communication and Synchronization
– how do the elements cooperate and communicate?
– how are data transmitted between processors?
– what are the abstractions and primitives for cooperation?
• Performance and Scalability
– how does it all translate into performance?
- how does it scale?

GPU and Grid

GPU-accelerated computing is the use of a graphics processing unit (GPU) together with a CPU to accelerate scientific, analytics, engineering, consumer, and enterprise applications. Pioneered in 2007 by NVIDIA, GPU accelerators now power energy-efficient datacenters in government labs, universities, enterprises, and small-and-medium businesses around the world. GPUs are accelerating applications in platforms ranging from cars, to mobile phones and tablets, to drones and robots.

HOW GPUS ACCELERATE APPLICATIONS

GPU-accelerated computing offers unprecedented application performance by offloading compute-intensive portions of the application to the GPU, while the remainder of the code still runs on the CPU. From a user's perspective, applications simply run significantly faster.

CPU VERSUS GPU

A simple way to understand the difference between a CPU and GPU is to compare how they process tasks. A CPU consists of a few cores optimized for sequential serial processing while a GPU has a massively parallel architecture consisting of thousands of smaller, more efficient cores designed for handling multiple tasks simultaneously.

GPUs have thousands of cores to process parallel workloads efficiently