Super-scalar Architecture: Evolution, Challenges and Considerations

1 Introduction

The emergence and evolution of super-scalar architecture has largely influenced the development of the computer field. In this report, we will explore it in three aspects. Firstly, we shall delve into the hardware part, tracing the evolution from single-core to multi-core processors and examining different types of multi-core processors and their developments over the years. Secondly, we will navigate through the operating system, getting to know about the main OSs capable of utilizing the power of multi-core processors, their evolution, and a comprehensive discussion of the features implemented by one OS: Linux. Finally, we will clarify the influence of two critical principles in parallel computing: Amdahl's Law and Gustafson's Law. These laws provide invaluable insights into the potential and limitations of parallel processing, shaping our understanding of super-scalar architecture's future.

2 The Rise of Multi-core Processors and PCs

2.1 Introduction to Multi-Core Processors and Their History

Multi-core processors are integrated circuits that contain two or more processing units, or cores. Each core can only operate one single instruction at the same time, however, due to the presence of multiple cores, the processor is able to execute several instructions simultaneously, thus increasing the overall speed on a macro level, resulting in faster and more efficient computing.

The concept of 'multi-core' was introduced in the 1980s as the improvement of clock speed of 'single-core' processors slowed. Multi-core processors gradually entered the commercial field in the early 21st century. The POWER4 Chip launched by IBM in 2001 was the first commercial multi-core processor (Chen *et al.*, 2009).

2.2 Types of Multi-Core Processors

2.2.1 Symmetric Multi-Processing (SMP)

In a SMP-type processor, there is typically one main memory which is shared by all processing units. Each core has equal access to reading from or writing to any memory location, as well as other resources like I/O devices, via system bus. Most multi-core processors in use today use SMP architecture. Fig. 1 shows the relationship between the different cores in SMP architecture.

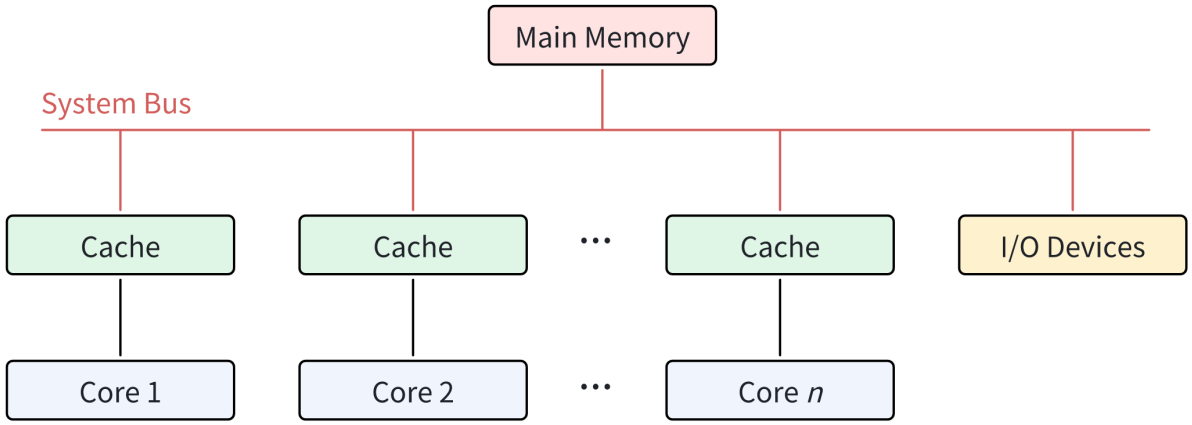

Fig. 1 Diagram of SMP Architecture

Advantages of SMP Architecture

(1) Affordability

Since there is only one shared main memory and one system bus for all the cores in SMP architecture, SMP-type processors are generally less expensive.

(2) Reliability

Since each core is relatively parallel in SMP architecture, even if there is a problem with one, the whole system is still able to operate at a reduced speed.

2.2.2 Non-Uniform Memory Access (NUMA)

The emergence of NUMA structure is later than that of SMP architecture, because it is a result of optimisation based on SMP architecture. In NUMA srchitecture, unlike SMP architecture, each core typically has its own local memory, or "near" memory, which the core can access with lower latency than other memory nodes, or "far" memory. Multiple cores may share one memory node. This compensates for the fact that memory operates significantly slower than processing units, which means that NUMA-type processors are able to provide better speed and performance than SMP-type processors in scenarios where multiple processing units need to access memory at the same time. NUMA architecture is prevalent in many high-performance computing environments nowadays, for instance, servers and workstations. Fig. 2 shows the relationship between the different cores in NUMA architecture.

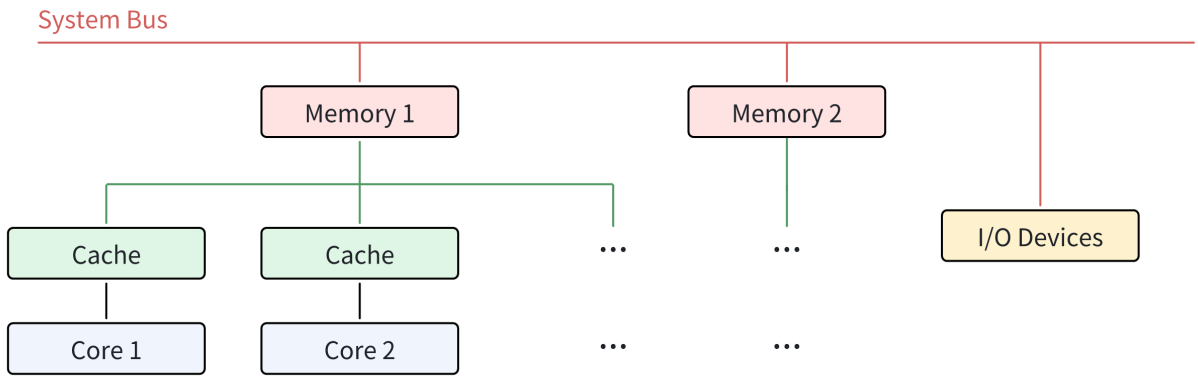

Fig. 2 Diagram of NUMA Architecture

Advantages of NUMA Architecture

(1) Performance

Since every core in NUMA architecture has its own local memory, the memory access times of cores are significantly reduced. This also means that the possible memory contention caused when multiple cores are accessing the memory simultaneously are avoided to the greatest extent, which improves the overall system performance.

(2) Scalability

NUMA architecture is built for heavy workloads. When more powerful performance is needed, additional cores and their memory nodes can be added to the processor. This proves that NUMA architecture has a strong scalability.

2.3 The Development of Multi-core Processors and Modern PCs

After multi-core processors entered the commercial field in the early 21th century, multi-core processors became widely used in consumer PCs. By the mid-2000s, dual-core processors became the mainstream in the PC market.

Around 2010, more advanced quad-core processors began to be adopted in consumer PCs. The advantages of multi-core processors are further reflected, because they will not significantly increase computing energy consumption while further improving the performance of computers.

The 2010s witnessed an explosion in the number of cores in commercial multi-core processors. Hexa-core, octa-core, and even higher-core processors emerged one after another. PCs were beginning to have the ability to perform tasks that were more complex and required more computing power than the past, for instance, running a large-scale video game. Ordinary people began to gain more powerful capabilities with the help of PCs.

Today, the design of multi-core processors begins to focus more on the optimisation for different cores, rather than simply accumulating the number of cores. Cores on every multi-core processors nowadays are becoming more specialized. Take Intel i9-13900H, one of the most powerful multi-core processors in PCs at present, as an example. Fig. 3 below (Amazon, 2023) outlines some of the parameters of the chip. As shown, Intel i9-13900H has 14 cores, of which 6 are performance cores, which are optimized for high-performance tasks, and the other 8 are energy-efficiency cores, which are specifically designed for efficiency. This kind of differential design allows PCs to be capable of a wide range of workloads.

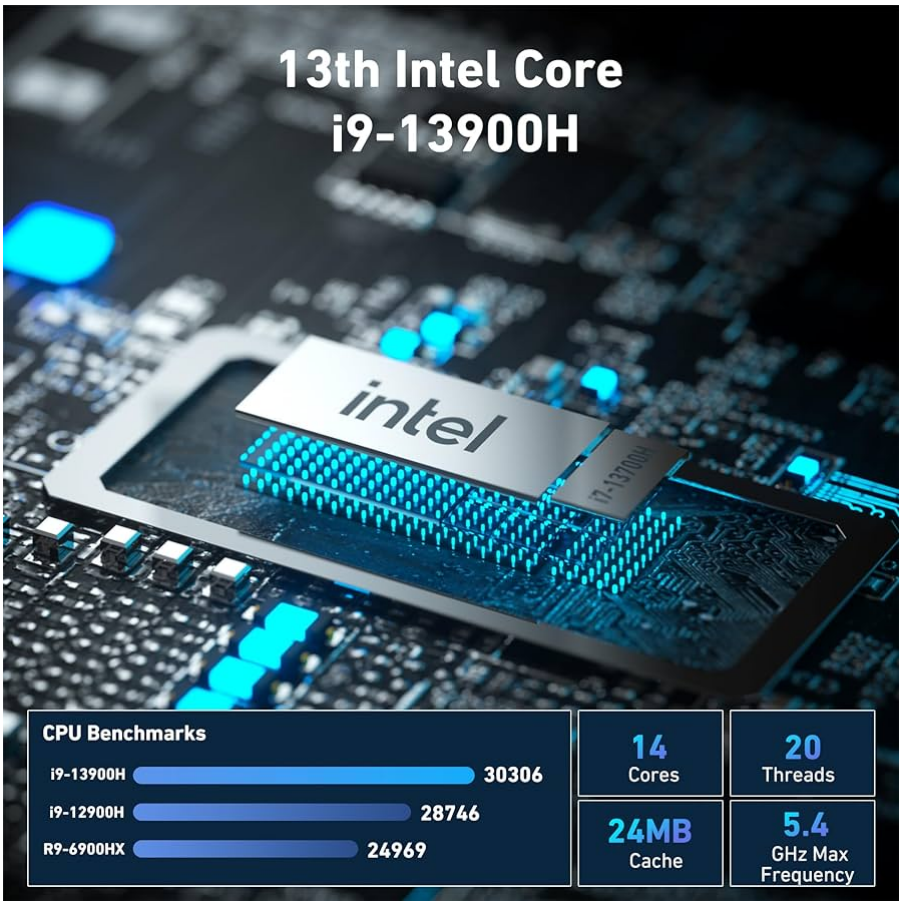

Fig. 3 Parameter Introduction of Intel i9-13900H (Amazon, 2023)

Since the appearance of multi-core processors, the adoption of them in PCs has significantly enhanced of capiability of modern PCs. The transformation of modern PCs has also led to the development of more sophisticated operating systems designed to make the most use of the parallel processing capabilities of multi-core processors. This will be covered in the next chapter.

3 Operating Systems

3.1 The Importance of Operating System Support for Multi-core Processors

(1) Resource Utilization

The operating system plays a important role in managing resources for different cores in a multi-core processor. Without the effective support of an operating system, the resources of a multi-core processor may not be properly utilized. The operating system is responsible for task scheduling, ensuring that each core is effectively utilized and that no core is idle while others are overloaded. This balance is critical in achieving high performance. Additionally, the operating system manages the memory hierarchy, ensuring that each core has the same access to memory resources, which is important for efficient execution of tasks.

(2) Scalability

As the number of cores in a processor increases, the complexity of managing these cores and their interactions also increases. The operating system must be capable of scaling its management capabilities to accommodate the increased number of cores. Without a scalable operating system, the benefits of adding more cores could be limited by the increased overhead of managing them, thereby limiting the potential performance improvements.

(3) Parallelism and Concurrency

Parallelism and concurrency are inherent advantages of multi-core processors, allowing multiple tasks to be executed simultaneously. However, this requires sophisticated support from operating system. It is responsible for managing concurrent tasks, ensuring that they are executed without conflicts and that the results are correctly synchronized. Furthermore, the operating system must also manage parallelism, dividing tasks into smaller subtasks that can be executed in parallel across multiple cores. This requires complex algorithms for task division and scheduling, as well as for managing communication and synchronization between cores.

3.2 The History and Development of Multi-core Utilization in Main OSs

Today, the most common mainstream operating systems in our lives, including Linux, Windows and macOS, are all well supported with multi-core processors.

(1) Linux

Linux began its support for SMP architecture in its version 2.0 in 1996. It introduced LinuxThreads as a partial implementation for POSIX threads, which is a thread standard for Portable Operating System Standard, or POSIX. With the introduction of POSIX threads, Linux was able to support user-level threading, enabling parallel task execution.

Later, more complete and efficient support for multi-core processing was further refined with subsequent releases. For instance, Linux version 2.6 provides better task scheduling and load balancing by introducing a new scheduling algorithm that is capable of scheduling different tasks in constant time.

(2) Windows

Windows OS began its support for SMP architecture in Windows NT 4.0 in 1996, but it was primarily designed for servers and workstations. Windows XP released in 2000 is the first Windows system to actually and directly support multi-core. It was optimized to take advantage of multi-core processors.

In 2010, Microsoft introduced .NET Framework 4.0, which brought an important new feature called Parallel Language Integrated Query, or PLINQ. PLINQ allows developers to write parallel queries in Language Integrated Query (LINQ). It also enabled automatic parallelization of certain LINQ queries. PLINQ is able to distribute the workload across multiple cores automatically, without requiring explicit threading code from the developer. This enables Windows to take full advantage of the parallel processing power of the multi-core architecture.

(3) macOS

The first version of macOS to officially support SMP was Mac OS X Server 1.0, released in 1999. Mac OS X Server 1.0 was based on the Mach kernel and BSD Unix, which provided a solid foundation for supporting SMP configurations. It offered true symmetric multiprocessing capabilities, allowing multiple cores to execute tasks concurrently.

In 2009, Apple released MacOS Snow Leopard. It introduced a feature called Grand Central Dispatch, or GCD. GCD is able to optimize application performance on multi-core processors and symmetric multiprocessing systems by giving more authority over threads to the operating system. With the help of GCD, developers can easily take advantage of multi-core processors without having to focus heavily on architecture of the operating system's thread pool.

3.3 A Detailed Description on How Linux Utilizes Multi-core Processors

Linux, as an open source OS, has a wealth of experience and flexible design in supporting multi-core processors, which is mainly based on its SMP architecture, which allows multiple cores to share the same memory space and the exact same I/O bus. In addition to this, there are also a number of features in Linux that further strengthens Linux's ability to utilize multiple cores:

3.3.1 Scheduling Algorithms: CFS (Completely Fair Scheduler)

CFS is the foundation of Linux's general task scheduling, focusing on time-sharing for non-real-time processes to ensure fair CPU allocation. It employs a virtual clock system, in which the advancement of each processor's clock slows with increased task weight. Moreover, each task's virtual clock is inversely proportional to its weight, effectively measuring CPU time (Liu *et al.*, 2010). CFS replaces the classic priority queue with a red-black tree for each CPU's run queue, optimizing multi-core scheduling, as shown in Fig. 4. High-priority tasks, though slower in virtual time, are scheduled often, promoting resource equity and preventing them from hogging the CPU at the expense of lower-priority tasks, thus improving system performance and responsiveness.

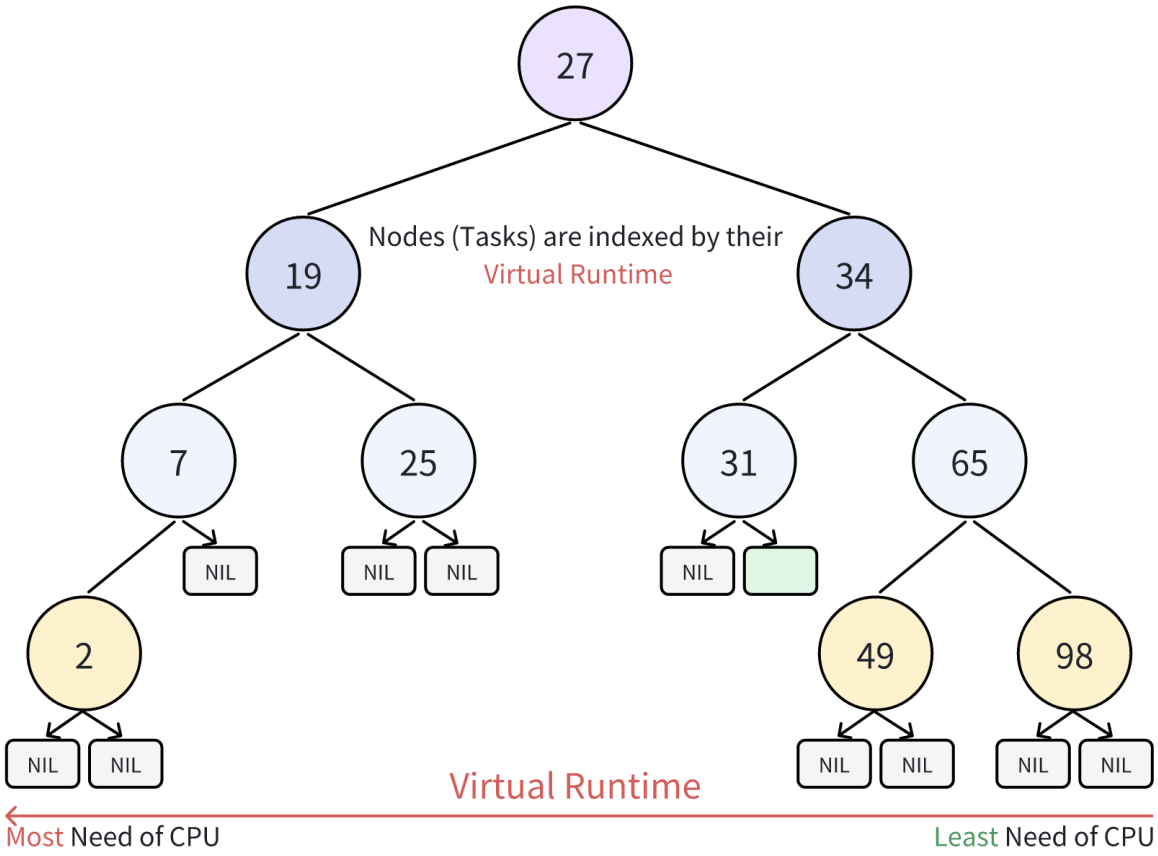

Fig. 4 The Red-Black Tree for Tasks in CFS

In addition to CFS, Linux offers several other scheduling policies, including:

- **SCHED_FIFO**: A FIFO real-time scheduling policy suitable for tasks that require strict timing guarantees.
- **SCHED_RR**: A Round-Robin realtime scheduling policy that allocates a fixed time slice to each task, ideal for tasks that require a higher priority but do not need to monopolize the CPU (Ishkov, 2015).
- **SCHED_OTHER**: A non-realtime time-sharing scheduling policy designed for regular user-level processes.
- **SCHED_IDLE**: A low-priority scheduling policy that runs only when no other higher priority tasks require the CPU.

3.3.2 Time Slicing

Linux employs CFS for time slicing, allocating CPU time to processes in a round-robin fashion. This ensures fair execution opportunities for all processes, enhancing system responsiveness and interactivity in a multitasking setting.

3.3.3 Thread-level Parallelism

As mentioned above, Linux provides a POSIX-standard kernel level multithreading library called LinuxThreads, which enables user programs to efficiently create, manage, and synchronize threads. These threads benefit from shared memory, which facilitates efficient communication and enhances application performance. Moreover, Linux supports CPU affinity, allowing for the optimization of thread execution on designated cores, reducing cache misses and inter-core communication costs, and effectively make use of the power of multi-core processors to improve overall system efficiency.

4 Challenges and Considerations in Multi-Core Processor Scalability

4.1 Limitations of Cores

Multi-core processors, with their ability to run multiple tasks simultaneously, have changed the field of computing. Each core in these processors perform as an individual, thus adding power for calculating. However, their rise has led issues like cache consistency, memory access patterns and inter-core communication.

Cache consistency is about making sure that changes to shared data are correctly updated for anyone who needs that. When a cache copy undergoes a memory writing operation, the cache controller invalidates the copy of data, suggesting a new value has to be fetched from the main memory in the next memory access (Park *et al.*, 1998).

Memory access latency is another issue. Even though processors are equipped with multiple instruction execution units, memory access causes cache misses, which slows down the program considerably. Fig. 5 indicated that the physical distances increases as system goes large.

The inter core communication also plays a crucial role in CPUs. The compiler should balance float and integer instructions, allowing the scheduling unit keeping all types of units busy. Better algorithm for compiler to optimally schedule instructions can lead to higher performance of CPU.

More cores do not always bring better performance. Overcoming these challenges is essential for true scalability when we increase the number of cores in a single processor.

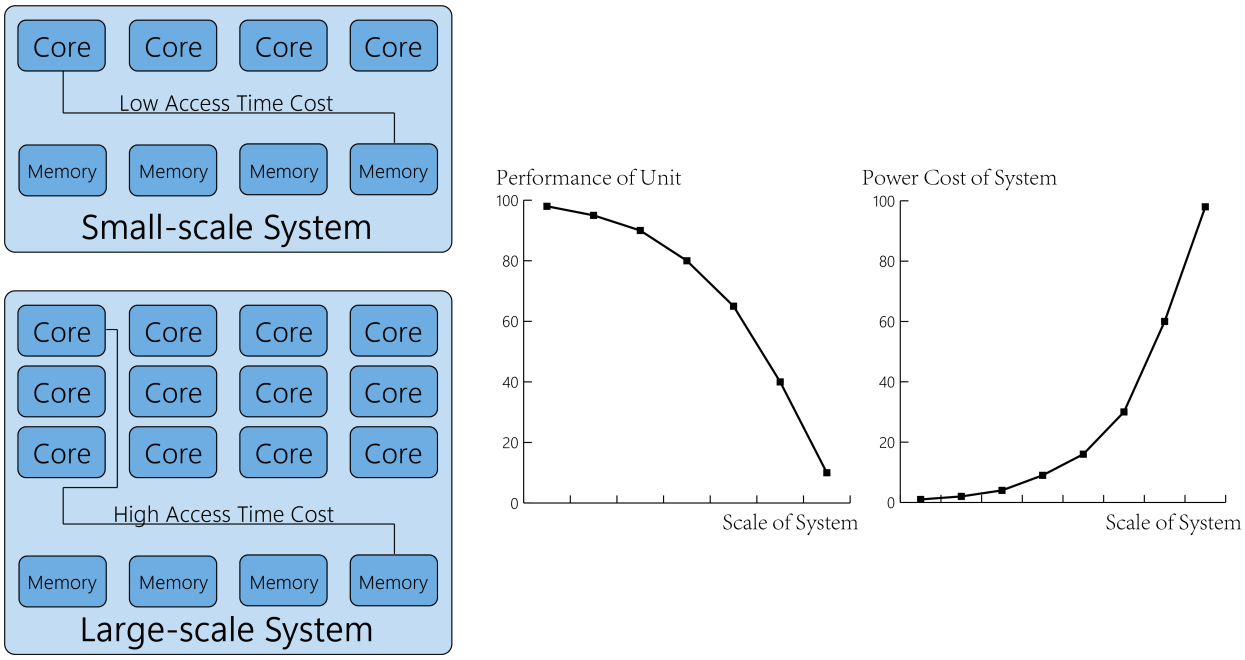

Fig. 5 Physical Delay of Large Systems

4.2 Limitations of Algorithms

Parallel programming and thread-level parallelism are of great importance to harnessing the capabilities of super-scalar architecture. Parallel programming allows concurrent computations, improving efficiency and speed.

On the other hand, thread-level parallelism involves different threads executing a single process simultaneously. This is possible benefit from the multiple execution units in a super-scalar processor, which can process separate threads concurrently, enhancing throughput and performance.

However, developing software that can effectively use these features is complex. Developers must consider synchronization, which coordinates the execution of multiple instructions. Poorly managed synchronization can create performance bottlenecks.

Load balancing, which involves distributing instructions evenly among functional units, is a vital aspect. It is essential to employ dynamic instruction dispatching algorithms to ensure a balanced computational load on units. Uneven distribution can result in either under-utilization or over-utilization of resources, finally impacting system performance. Fig. 6 gives a brief sketch of how tasks were distributed to cores.

Additionally, effective management of competition for shared resources such as registers and caches is crucial, too. In super-scalar architecture, multiple execution units compete for these shared resources, potentially causing bottlenecks of calculating strength.

Although parallel programming and thread-level parallelism offer significant advantages, they also come with distinct challenges. Successfully addressing these challenges necessitates meticulous handling of synchronization, load balancing, and resource contention.

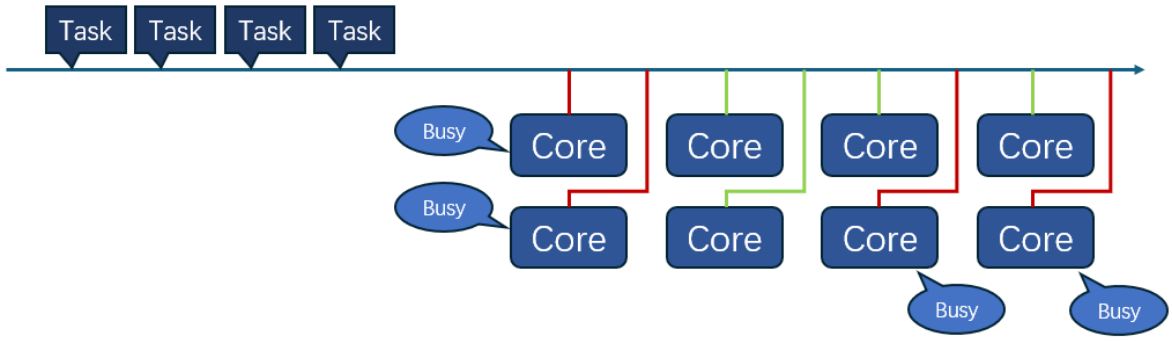

Fig. 6 Tasks Distributed to Cores Sketch

4.3 Trade-offs on Designing

In the context of super-scalar architecture, there are several trade-offs to consider. One of the key principles is Amdahl's law, which states that even if parallelized, some parts of the application are still continuous. Amdahl's law assumes that the speed-up ratio of parallelization is limited by the part of the program that cannot be parallelized (Wu *et al.*, 2011). This means that no matter how many cores are added to the processor, some programs cannot take advantage of this parallelism, which limits the overall acceleration that can be achieved.

On the other hand, Gustafson's Law highlights that as the size of a problem grows, the significance of the parallel aspect also increases to enhance scalability. According to this principle, with the availability of more computing resources, they are typically allocated to tackle larger problems. In this scenario, the time spent on the parallelizable segment is often considerably faster than the inherent serial tasks.

This means that designing better method and algorithm to realize functions and turn sequential computing into parallel computing will also help improve scalability. Interestingly, slower algorithms that can run in parallel sometimes work better on faster sequential algorithms on multi-core processors.

Finding the right balance is not easy. It's important to handle parallel tasks, synchronization, and communication overhead carefully. In super-scalar processors, problems like delays become more obvious and can affect performance. This can lead to scheduling issues. So, when creating super-scalar architecture, it's crucial to think about these factors carefully.

5 Conclusion

The evolution of multi-core processors and multi-processor PCs has revolutionized the hardware landscape, while advancements in operating systems have adapted to harness this increased processing power, illustrating the symbiotic relationship between hardware and software in the journey towards more efficient computing.

The future of computing may be dominated by multi-core processors, with more and more emphasis on hardware and software level parallelism. There are undoubtedly still many challenges in this field, but we can still look forward to the future development of super-scalar architectures.

6 References

Amazon (2023) *GEEKOM Mini PC Mini IT13, 13th Gen Intel i9-13900H NUC13 Mini Computers(14 Cores,20 Threads) 32GB DDR4 & 2TB PCIe Gen 4 SSD Windows 11 Pro Desktop PC Support Wi-Fi 6E/Bluetooth 5.2/USB 4.0/2.5G LAN/8K*. Available at: https://www.amazon.com/GEEKOM-Mini-IT13-i9-13900H-Computers/dp/B0CJTGHNV2 (Accessed: 13 April 2024).

Chen, G.L., Sun, G.Z., Xu, Y. and Long, B. (2009) 'Integrated research of parallel computing: Status and future', *Science Bulletin*, 54(11), pp. 1845-1853. Available at: https://link.springer.com/article/10.1007/s11434-009-0261-9. (Accessed: 14 April 2024).

Ishkov, N. (2015) 'A complete guide to Linux process scheduling (Master's thesis)', *M.Sc. Thesis*. Available at: https://trepo.tuni.fi/bitstream/handle/10024/96864/GRADU-1428493916.pdf?sequence=1&isAllowed=y (Accessed: 15 April 2024).

Liu, T., Wang, H.J., Wang, G.H. (2010) 'Research of CFS Scheduling Algorithm Based on Linux Kernel', *Computer & Telecommunication*, 4(3), pp. 61-63. Available at: https://qikan.cqvip.com/Qikan/Article/Detail?id=33544743&from=Qikan_Search_Index (Accessed: 16 April 2024).

Park, G.H., Kwon, O.Y., Han T.D., Kim, S.D. and Yang S.B. (1998) 'Methods to improve performance of instruction prefetching through balanced improvement of two primary performance factors', *Journal of Systems Architecture*, 1998(9), pp. 755-772. (Accessed: 17 April 2024).

Wu, X.L., Beissinger, T.M., Bauck S., Woodward, B., Rosa G.J.M., Weigel, K.A., Gatti, N.L. and Gianola, D. (2011) 'A Primer on High-Throughput Computing for Genomic Selection', *Frontiers in Genetics*. (Accessed: 16 April 2024).