Hardware and Software Strategies for Parallelism and Distributed Processing

Loic Niragire

School of Technology, Northcentral University

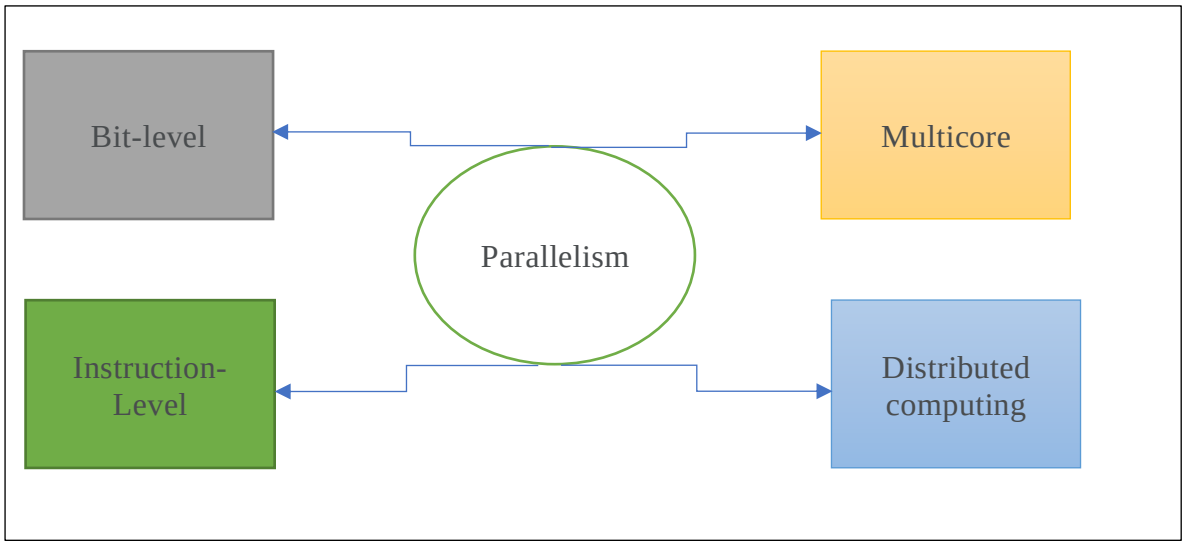
TIM 8121: Distributed Algorithms and Parallel Computing

Dr. Carol Cusano

May 21st, 2022

This is a comparative analysis of the performance impact of various parallelism types applied on a video rendering system that seeks to render 10 video files each at 1GB. By maintaining the same clock rate throughout, each system configuration is compared against an established benchmark where a single file takes 128 seconds to render on a 32-bit scalar processor clocked at 3.5GHz. The first case seeks to determine the execution time from applying bit-level parallelism and increasing the word length from 32-bit to 64-bit. Following that, the second case introduces instruction pipelining. Then the processor is upgraded to a quad-core without pipelining support in the third case. Finally, the system is augmented to a 5-node distributed system on a 100 Mbps network. A comprehensive strategy for solid hardware and software architectures must incorporate a combination of parallelism types to support concurrent systems because individual parallelism types are only optimized for specific scenarios. Figure 1 depicts the parallelism types evaluated in this paper.

Figure 1 Parallelism types



Analysis

The performance of a computer is determined by three factors: instruction count, clock cycle time, and clock cycles per instruction. The instruction count is depended on the programming language, algorithms, compiler, and instruction set architecture. The clock cycle time and the number of clock cycles per instruction depend on the processor's implementation (Patterson & Hennessy, 2014).

Using a scalar processor, the analysis performed in this paper focuses on the effect of different parallelism types on CPU execution time. A scalar processor, by definition, acts on one piece of data at a time – one instruction per clock cycle. As shown in Equation 1, CPU execution time can be improved by either increasing the clock rate or reducing clock cycles. Note that the clock rate, 3.5 GHz, used throughout this analysis remains fixed.

$$CPU\ execution\ time = \frac{CPU\ clock\ cycles}{Clock\ rate} \qquad \text{Eq. (1)}$$

Therefore, the CPU clock cycles associated with the benchmark case is computing as:

$$128\ sec = \frac{CPU\ clock\ cycles}{3.5 \text{ x } 10^9\ {}^{cycles}/_{sec}}$$

$$CPU\ clock\ cycles = 128\ \text{sec x } 3.5\text{x}10^9\ {}^{cycles}/_{sec} = 4.48 \text{ x } 10^{11}\ cycles \qquad \text{Eq. (2)}$$

The CPU clock cycles depend on the number of instructions, as illustrated in Equation 3. Clock cycles per instruction refers to the average number of clock cycles each instruction takes to execute – also known as CPI. In other words, executing the same number of instructions may result in different CPU clock cycles depending on the instruction set implementation. Thus, the CPI provides a way to compare different implementations of the

same instruction set architecture (Patterson & Hennessy, 2014). Instructions per clock cycle is another measure often used to derive CPI. A processor that executes x number of instructions per clock cycle, has a CPI of $1/x$. Therefore, a scalar processor has a CPI of 1 since it executes one instruction per clock cycle.

$$CPU\ clock\ cycles = instruction\ count \text{ x Average clock clycles per instruction} \quad \text{Eq (3)}$$

Video rendering consists of five sub-tasks; reading footage, decoding footage, adding effects, compressing footage, and saving final output file (Harper, 2022). Notice that three out of five sub-tasks are CPU-bound. Therefore, according to Amdahl's law, only two out of five video rendering sub-tasks will not benefit from any parallelization. Thus, the speedup calculated throughout this analysis only reflect improvements in CPU-bound sub-tasks.

Table 1 Video rendering sub-tasks

Sub-task	*Impacted by*
Reading footage	Storage read speed and RAM capacity
Decoding footage	CPU or GPU
Adding effects	CPU or GPU
Compress footage	CPU or GPU
Save final output file	Storage speed

Bit-level Parallelism

The first case analyzed explores bit-level parallelism by increasing the word length from 32-bit to 64-bit. Therefore, reducing the clock cycle count by addressing 2^{64} bits of

RAM in a single clock cycle. According to Equation 3, the instruction count or the average clock cycles per instruction (CPI) must be reduced to reduce the CPU clock cycles.

Let *n* be the clock cycle count factor from switching to a 64-bit processor, then Eq. 4 shows the corresponding CPU execution time. To calculate the CPU execution time from bit-level parallelism, the following assumptions are made:

- Negligible impact on the *instruction count* from the programming language, compiler, and algorithms choice.
- The source code is recompiled for a 64-bit processor.
- The program is executed on a 64-bit operating system to maximize added memory access.
- A 10% reduction in executed instruction count from efficient ISA

Equation 4 shows the resulting CPU execution time by applying bit-level parallelism.

$$\text{CPU execution time} = \frac{4.48\text{x}10^{11}\text{cycles} - (\text{n x } 4.48\text{x}10^{11}\text{cycles})}{3.5\text{x}10^{9}\,{}^{\text{cycles}}/_{\text{sec}}} \qquad \text{Eq (4)}$$

$$\text{let n} = 0.1$$

$$\text{CPU execution time} = 115.2\ \text{sec}$$

$$\text{bit} - \text{level speedup} = \frac{128\ \text{sec}}{115.2\ \text{sec}} = 1.11 \qquad \text{Eq (5)}$$

Instruction Pipelining

Pipelining has a speed-up of approximately the number of stages, thus increasing the depth of the pipeline improves the potential of instruction parallelism (Patterson & Hennessy, 2014). The calculations below assume a five-stage pipeline that can execute four instructions

per clock cycle without a register delay. If we let *n* equal the number of stages in the pipeline, and *k* be the CPU clock cycles, then the total pipeline execution time is given by Equation 6 below. By using the clock cycles value obtained from Equation 2, the pipeline execution time is shown in below.

$$\text{Execution time } = (n + k - 1)\ \frac{1}{clock\ rate} \qquad \text{Eq (6)}$$

With four instructions per clock cycle, execution time = $\left(n + \frac{k}{4} - 1\right)\frac{1}{clock\ rate}$

Thus, CPU execution time = $\frac{1}{3.5\ GHz}$ $(5 + 1.12\text{x}10^{11} - 1)$ = 32 sec

$$ILP\ speedup = \frac{128\ sec}{32\ sec} = 4$$

Multicore Processing

The following parallelization upgrades the benchmark CPU from a 3.5 GHz 32-bit single-core to a 3.5 GHz 32-bit quad-core processor without pipelining. Assuming that each core will share the video rendering load equally. To obtain the execution time, let *c* represent the overhead cost of splitting *k* tasks across *n* processors. Suppose *c* = 1, then Equation 7 gives the execution time model.

$$Execution\ time = \left(128\ sec * \frac{10}{4}\right) + (\frac{k}{n} + c) \qquad \text{Eq (7)}$$

$$For\ n = 1, execution\ time = 128\ sec, k = 127$$

$$Therefore, for\ n = 4, execution\ time = 320\,\text{sec} + \ (^{127}/_{4} + c)$$

$$= 323.75\ seconds$$

$$Multicore\ speedup = \frac{1280sec}{323.75sec} = 3.95$$

Distributed Computing

This model upgrades the benchmark case from a single 3.5 GHz 32-bit processor to five distributed 32-bit processors at 3.5 GHz on a 100 Mb/s network speed. By dividing the amount of work evenly across processors, each processor will render 10/5 video files. Additionally, a network delay needs to be added to the processing time of each video file to account for the upload time. Recall that 1byte = 8 bits and 1GB = 1024 MB, then 100 Mbps equals approximately 0.0125 GB/s. Moreover, we need to calculate the time it will take to transfer each video file across the network to the processing CPU. Since the video file is 1GB in size and the network speed is 0.0125GB/s, the time to transfer one file would be 80 seconds. But we must remember that we have to transfer the file twice: once to upload it to the CPU for processing and once to download it after it's rendered. Therefore, the total transfer time per file is 160 seconds. Equation 8 gives the execution time for each processor to upload and render all 10 video files.

$$\text{Execution time} = (160\,\text{sec} + 128\,\text{sec}) * {}^{10}/_{5} = 576\,sec \qquad \text{Eq (8)}$$

$$Distributed\ computing\ speedup = \frac{1280sec}{576sec} = 2.22$$

Observation

Table 2 highlights the calculated execution times for rendering all 10 videos using various parallelism types. The best parallelization model for a concurrent system to render 10 video files is the distributed model with five 32-bit processors. Although the calculated execution time for this model is not the lowest, it makes the least assumptions and avoids data dependency issues. The multicore model facilitates true parallelism by scheduling threads on

separate cores but has an unknown data synchronization cost due to the threads' shared memory model. Moreover, converting an application from a single-thread model to a multi-threaded one requires significant effort. Most notably, distributed computing is more scalable and only limited by the network speed, which is easier to upgrade without modifying source code. For instance, the amount of parallelization available via the distributed model can be maximized by upgrading some or all distributed nodes to 64-bit multi-core processors with instruction pipelining support.

The single-core CPU with instruction pipelining support yielded better execution time than the distributed model. Although several instructions are designed for pipelining, it takes a coordinated effort between the hardware and software to avoid pipeline hazards. Additionally, the execution time obtained using this model is simplified by assuming that each instruction takes an equal amount of clock cycles. Moreover, it does not account for stage and register delays.

Increasing the word length from 32-bit to 64-bit resulted in the least parallelization because it seeks to reduce executed instruction count in a single process. Since a 32-bit processor has an address space of 4GB, the added address spaces from updating to a 64-bit process are unnecessary to process a 1GB video file. Moreover, a 10% instruction count reduction is assumed from increasing the word length to 64-bit without accounting for CPU idle time due to I/O activities.

Table 2 Parallelism execution times

CASES	CPU ARCHITECTURE		EXECUTION TIME
BENCHMARK	32-bit 3.5 GHz	Single-core scalar CPU	1280 seconds
1	64-bit 3.5 GHz	Single-core scalar CPU	1150.2 sec
2	32-bit 3.5 GHz	Single-core scalar CPU with instruction pipelining	320 sec
3	32-bit 3.5 GHz	Quad-core scalar CPU without pipelining	323.75 sec
4	Five 32-bit 3.5 GHz	- Single-core scalar CPU - Network upload/download speed of 100 Mbps	576 sec

Architecture Strategy

Combining different parallelization types yields the most parallelization in a concurrent system. The decision of the types of parallelization to consider should be evaluated based on the system's overall goal and available resources. The goals can be categorized into two groups. Either the system aims to improve throughput or increase execution time. Systems that aim to improve throughput are more concerned with processing independent tasks. Meanwhile, those that aim to increase execution time are typically focused on solving big problems that cannot fit on a single machine. The scenario analyzed in this paper aims to improve the systems' throughput of processing 10 independent tasks.

Distributed computing fits the first category, where the goal is to increase the overall throughput of processing a set of independent tasks. Although it introduces a network component and multiple failure points, the distributed computing model provides the most fault tolerance (Fraigniaud, 2013). To process 10 independent tasks, for instance, a 10x speedup can be realized by running the jobs on a 10-node distributed system.

Further parallelization can be achieved by upgrading nodes' architecture in a distributed system. For instance, processors with multiple processing units can enhance potential parallelism through hardware duplication. Using our video rendering system as an example, we observed that processing a single video file on a quad-core processor resulted in 3.95 speedup compared to running it on a single-core processor of the same clock speed.

The Single Instruction, Multiple Data (SIMD) architecture supports multicore processing where a single application is executed across all systems' cores. The OS, however, perceives each core as a separate processor. The main disadvantage of using this architecture style is that the cores have shared memory, thus requiring data protection mechanisms to prevent threads from prematurely overwriting shared data.

Multiple Instructions, Multiple Data (MIMD) is another architecture style that supports hardware duplication by giving each processor its memory. Unlike the SIMD model, processors in MIMD function asynchronously and communicate through message passing. Each processor in MIMD model has a separate program with an instruction stream (JavaTPoint, 2023).

To maximize core utilization, instruction-level parallelism (ILP) attempts to reduce CPU idle time by rearranging the instruction's execution sequence within a thread of execution. The CPU achieves ILPU through software and hardware techniques such as

superscalar, register renaming, out-of-order execution, speculative execution, multiple-issue, VLIW, and dynamic scheduling. The effectiveness of ILP is limited by data dependencies and the ability of the pipeline to avoid hazards, which occurs when the next instruction cannot be executed in the following clock cycle.

Three types of dependencies can introduce stalls in ILP, where a stall is defined as a cycle in the pipeline without a new input (GeeksforGeeks, 2023). A structural dependency occurs when multiple instructions attempt to access the same resource in the same cycle, resulting in a pipeline stall until the resource becomes available. Meanwhile, a control dependency occurs when the processor fails to determine the branching target address, resulting in a branch penalty. Lastly, there is data dependency, also known as data hazard. It occurs when a pipeline stage depends on data not yet available.

The potential ILP can be enhanced by increasing the pipeline depth or issuing multiple instructions at the beginning of each clock cycle – multiple-issue. Today's high-end CPU attempts to issue from 3 to 6 instructions in every clock cycle (Patterson & Hennessy, 2014). Table 3 illustrates a 2-issue static pipeline with 5 stages – instruction fetch (IF), instruction decode (ID), execution (EX), data memory access (MEM), and write back (WB). One of ILP's main challenges involves scheduling instructions in the right sequence and cycle. To this end, a static multiple-issue CPU uses the compiler to assist with packaging instructions and handling hazards. A dynamic multiple-issue CPU, or superscalar processor, decides how many instructions to issue in each clock cycle.

Table 3 Static 2-issue pipeline with 5 stages

Instruction type	*Pipeline Stages*						
ALU	IF	ID	EX	MEM	WB		
LOAD	IF	ID	EX	MEM	WB		
ALU		IF	ID	EX	MEM	WB	
LOAD		IF	ID	EX	MEM	WB	
ALU			IF	ID	EX	MEM	WB
LOAD			IF	ID	EX	MEM	WB

The Bit-level parallelism case assumes a 10% reduction in instruction count. However, the exact degree of reduction in instruction count depends on several factors, such as the nature of the ISA, the specific instruction optimizations available in the ISA, the quality of the compiler, and the nature of the task being performed. Although various execution speeds could be obtained using different assumptions, combining all four parallelization techniques will result in the highest parallelization. Distributed computing provided the best parallelization for the video rending concurrent system analyzed since the tasks are independent. Additionally, distributed computing provides horizontal scalability, which improves availability and resilience. Further parallelization can be realized within a distributed system by introducing additional hardware capabilities. Although multicore processing provides the second-best parallelization, it requires source changes and has dependency on the operating system, runtime environment, and algorithms choice.

Conversely, ILP provides equally good improvements without much code change since it's mostly hardware dependent. Bit-level parallelism provided the least improvement.

References

Fraigniaud, P. K. (2013). Towards a Complexity Theory for Local Distributed Computing. *Proceedings of the 25th IEEE Symposium on Foundations of Computer Sciece (FOCS)* (pp. 35-61). New York: IEEE.

GeeksforGeeks. (2023, January 10). *Computer Organization and Architecture.* Retrieved from geeksforgeeks.org: https://www.geeksforgeeks.org/computer-organization-and-architecture-pipelining-set-2-dependencies-and-data-hazard/

Harper, C. (2022, July 12). Retrieved from CGDirector: https://www.cgdirector.com/how-long-to-render-a-video/

JavaTPoint. (2023, January 10). *Types of Pipeline Delay and Stalling.* Retrieved from Java TPoint: https://www.javatpoint.com/types-of-pipeline-delay-and-stalling

Patterson, A. D., & Hennessy, L. J. (2014). *Computer Organization and Design: the hardware/software interface.* Waltham: Morgan Kaufmann.