Santhosh Ramachandran

Assignment 6 – Thread Level Parallelism

Dr. Lively

Student ID: 005027659

Thread Level Parallelism (TLP) is a type of parallelism in which a system or processor can execute operations in multiple threads simultaneously. TLP is achieved by running different types of independent tasks or even subparts on distinct threads, saving much time. This multi-threading can happen across cores or even processors.

**Historical development of Thread Level Parallelism**

TLP has seen significant shifts and innovations in the design and utilization of microprocessors over the years. As (Otellini, 2005) mentioned: “We are dedicating all our future product development to multicore designs. We believe this is a key inflection point for the industry.” In 1986–2003, uniprocessor performance growth, driven by the microprocessor, was at its peak since the first transistorized computers in the late 1950s. Despite that, the importance of multiprocessors was constantly growing throughout the 1990s as designers figured out a way to build servers and supercomputers that achieved better performance than a single microprocessor. As time evolved, the returns were diminishing with Instruction level parallelism, causing the slowdown in the uniprocessor. This led to a new era in computer architecture where multiprocessors play a significant role, from small computers to supercomputers. Some key factors that led to the increased importance of multiprocessing include growth in data-intensive applications driven by the availability of massive amounts of data on the Internet and the interest and significance of high-end servers such as cloud computing and software-as-a-service.

Below is a chart displaying the development of TLP, highlighting the key milestones.

A screen shot of a graph

Description automatically generated

In the 1980s, the concept of multithreading emerged. They were allowing a single program to run multiple threads. When this was introduced, an explicit threading model was followed. This required the developers or creators to handle thread creation, sync, and maintenance manually. Later, in the 1990s, Symmetric Multiprocessing (SMP) was introduced. Architectures built with SMP utilized multiple CPUs that shared the same memory. This provided significant support to aid TLP and an improved parallel execution. At this phase, Thread Management APIs were becoming more and more common, but programmers still had to maintain the threads' synchronization mechanisms manually. This was also around the same time when dual-core and multi-core processors were beginning to become prevalent. This provided a way to enable hardware-level multithreaded execution, significantly boosting the performance of thread-level parallelism. Later, in the 2000-2010 era, processors transitioned from single-core to multi-core architectures and were able to run different threads on different cores. Also, at this time, dual-core and quad-core processors became the mainstream and were later followed by many-core designs. In this era, programming models used to maintain the threads have significantly improved, making programmers' lives a lot easier. Java's ForkJoinPool and Python's concurrent.futures were used to handle the threads.

**Parallelism models**

In TLP, we can leverage multiple parallelism models to achieve efficient parallelism. Starting with a shared memory model. In a shared memory model, multiple threads or processes share the same memory space and can communicate by reading and writing to shared variables. This is majorly effective and used for communication between threads. Secondly, the message-passing model is the idea of threads having their own local memory. Like a client and server, communication happens by sending and receiving messages. Moreover, we also have task parallelism in which different threads perform different tasks that may or may not be dependent on each other. This is done using task scheduling frameworks.

**Synchronization**

Synchronization can become a performance bottleneck since the contention introduces multiple delays and higher latency in multiprocessors. One of the everyday synchronization operations is atomic exchange. This operation interchanges a value in the register for a value in the memory. However, the nuclear operation has some challenges. Single atomic memory operation requires both a memory read and a write in a single instruction. Because of this expectation, coherence implementation becomes complicated. Hardware will not allow any other operation between the read and the write. All this should be taken into consideration while not allowing a deadlock. For example, when executing the below instruction, if the store conditional operation fails, caution should be taken when choosing what instructions should be inserted. In particular, only register-register instructions should be allowed. If this is not followed, it is possible to create a deadlock where the processor can never complete SC.

**Load Balancing**

**Static Load Balancing:** The workload is divided among threads before execution begins. This works well when the workload is predictable but can be inefficient if the tasks vary significantly in their computational demands. Imagine assigning tasks to your team members at the start of a project.

**Dynamic Load Balancing:** The processor adjusts thread assignments during runtime based on factors like core utilization and task priorities. This is more flexible but adds overhead for monitoring and reassignment.

**Performance Metrics**

-       **Speedup:** This metric measures the improvement in execution time when a program is run in parallel compared to sequential execution. Higher speedup indicates better TLP. However, it can be limited by factors like Amdahl's Law, which states that the non-parallelizable part of the program sets an upper bound on potential speedup.

-       **Throughput** refers to the number of tasks or operations a system can process in a given period. It's a good measure of how well parallel resources are utilized, especially in systems that handle many concurrent tasks.

-       **Scalability**: This indicates how effectively the performance of a system improves when more parallel resources (e.g., threads or cores) are added. Scalability can be affected by resource contention, synchronization overhead, and communication delays.

-       **Latency**: While parallelization can reduce the execution time, the time taken for individual operations or responses (latency) might increase if the workload involves significant synchronization or communication between threads.

**Critique Current Challenges**

**Scalability and Amdahl's Law**

The following screenshot shows the scalability of different workloads in an Xeon MP.

A graph with a line graph

Description automatically generated with medium confidence

* SPECjbb2015: This represents an IT system that handles point-of-sale requests, online purchases, and so on.
* SPECVirt2013: This represents a collection of VMs running independent combinations of other SPEC benchmarks.
* SPECOMP2012: This represents a collection of 14 scientific and engineering programs written with the OpenMP standard for shared-memory systems.

As we can see from the graph, the minor configuration for which these benchmarks vary from 48 to 72 cores. These different workloads signify the scalability of multicores in multiprocessors. Multicore processors offer significant performance benefits with TLP. On top of it, scalability depends heavily on factors such as workload characteristics, memory access patterns, cache coherence, and synchronization overhead. Software developers should consider how to maximize the utilization of available multicores and attain thread-level parallelism. This hardware-level knowledge should only be known to developers who primarily work on building high computation-intensive operations**.**

(Che, H., Nguyen, M., 2014) have provided an excellent analysis of how Amdahl’s law applies to multithreaded multicore processors. They developed a new thread-level closed-queuing network model, covering various design factors such as different types of cores (coarse-grain, fine-grain, SMT). They shared resources like cache, memory, and critical sections. A closed-form solution is derived for measuring speedup, allowing comprehensive analysis of MMP performance scaling.

The analysis reveals that for the parallelizable portion of a workload, the speedup is influenced not just by the number of processing units, as Amdahl’s law suggests, but also by workload characteristics, which may range from memory-bound to CPU-bound. Notably, the study finds that multithreading can produce super-linear speedup for memory-bound workloads with strong cache affinity. A tight upper bound on speedup is also derived, showing that resource contention, whether from shared memory or critical sections, introduces a sequential component in the parallelizable workload, limiting scalability. This adds to the inherent sequential constraint posed by Amdahl’s law.

To boost MMP performance, enhancing memory parallelism and minimizing critical sections to as few threads as possible within the same core is recommended.

**Heterogeneity, Energy efficiency & Parallelism models**

(S. K. Gutiérrez *et al*. 2017) Has published a paper on how Thread-Level Heterogeneity can be accommodated in Coupled Parallel Applications. This paper has discussed how Hybrid parallel programming models that merge message passing (MP) with multithreading (MT) are increasingly popular as they extend traditional MP, which typically relies on single-threaded processes for intra- and inter-node parallelism. This trend leads to more complex parallel applications that blend MP and MP+MT libraries with varying levels of threading, creating thread-level heterogeneity. Aligning the threading levels of these independently developed libraries is complex, and static resource binding by job launchers complicates the issue further.

A standard solution is to under-utilize compute resources so that the library with the highest thread count per process gets one processing element per thread. However, this often results in less use of resources when other libraries use fewer threads. To address this, they proposed a new approach that maximizes resource usage throughout the application by dynamically reconfiguring runtime environments based on threading and memory needs during different compute phases. Our method has demonstrated up to a 5.8× performance boost in real-world production codes, validated by over a year of continuous use and adoption in several production systems. They performed and proved these results on a model named QUO; this is both a model and an implementation that facilitates varying requirements of different computation phases in coupled MP+MT applications. Here is the QUO architecture diagram:

A diagram of several different types of applications

Description automatically generated

Also, the plot of average execution times of QUO operations. These operations were performed on cray XE6.

A graph of a graph with numbers and symbols

Description automatically generated with medium confidence

**Energy efficiency**

(Gebhart, M. et al., 2011) Has provided a comprehensive analysis of the energy efficiency of managing threads in throughput processors, especially GPUs. Modern GPUs use many hardware threads to mask delays in function units and memory access. Still, this extreme multithreading demands complex scheduling and a large register file, which can be energy-intensive and slow to access. They introduced two techniques to cut energy use in GPUs. First, they proposed register file caching, which reduces access to the central register file by using a smaller cache for the immediate working set of active threads. Second, they explored a two-level thread scheduler that keeps a limited number of active threads to handle ALU and local memory delays while a larger pool of pending threads handles main memory delays. These techniques optimize energy use by only allocating temporary register cache resources to currently active threads.

Their findings show that using a 6-entry per-thread register file cache can cut main register file reads and writes by 50% and 59%, respectively. Additionally, reducing the active thread count by fourfold has minimal performance impact and leads to a 36% reduction in register file energy. These improvements are validated across various real-world graphics and computing tasks.

**How challenges in TLP are addressed**

**Compiler Optimization**

(Yiapanis, P., et al., 2015) Has provided a detailed overview and results on how compiler-driven software speculation can benefit TLP. Static compiler methods can often parallelize code effectively but struggle when information needed for parallelization is only available at runtime. Early approaches to runtime parallelism used the Inspector/Executor model, where the inspector phase determined if a loop could run in parallel, followed by an executor phase that carried it out. However, the inspection overhead limited its practicality, leading researchers to focus on speculative parallelization, also known as thread-level speculation (TLS).

Speculative parallelization runs code in parallel optimistically, without knowing in advance if it can be safely parallelized and includes mechanisms to ensure correctness during execution. This involves scheduling speculative threads, monitoring their memory accesses, detecting unwanted actions, and preventing incorrect data from being stored in the main memory.

**Cache Coherence**

Cache coherence is a critical concept in computer architecture, especially in systems with multiple processors or cores, each having its own cache memory. It ensures that all caches have a consistent view of shared data, even when updates are made. Cache coherence protocols are implemented in hardware to maintain consistency. These protocols track which caches have copies of a particular data item and ensure that updates made by one core are propagated to other cores. This is typically achieved through mechanisms like Snooping and Directory-based. Moreover, (Singh, I. et al., 2013) mentioned While scalable coherence has been well-researched for general-purpose chip multiprocessors (CMPs), applying these methods to GPU architectures poses new challenges. Conventional directory protocols can lead to unnecessary coherence traffic and complicate the verification of GPU memory systems. Recent studies, like Library Cache Coherence (LCC), have explored time-based strategies for CMPs. They have introduced a time-based coherence framework for GPUs called Temporal Coherence (TC), which leverages globally synchronized counters in single-chip systems to create an efficient GPU coherence protocol. These counters allow all cache coherence operations, such as invalidating cache blocks, to occur synchronously, eliminating coherence traffic and protocol race conditions.

An implementation of TC, TC-Weak, is described, which avoids LCC's trade-offs between stalling store operations and increasing L1 cache miss rates. This approach enhances performance and reduces interconnect traffic by providing coherent L1 caches. TC-Weak demonstrates an 85% performance boost in GPU applications requiring inter-workgroup communication compared to turning off non-coherent L1 caches in standard GPUs. The study also highlights that write-through protocols are better suited for GPUs than writeback protocols, which can create excess traffic due to redundant refills of write-once data.

**Synthesize Future Directions**

**Heterogeneous Processing**

We're moving beyond CPUs with identical cores. Future systems will likely feature specialized cores for different tasks (e.g., AI acceleration, graphics processing). This requires sophisticated TLP techniques to manage and schedule threads across diverse core types efficiently. Think of it like a construction crew: you wouldn't use a plumber to do electrical work. Similarly, specialized cores optimize performance.

**Compiler-Assisted Threading**

Compilers are becoming smarter at automatically identifying and exploiting parallelism in programs. This relieves programmers from the burden of manual thread management, making TLP more accessible and efficient.

**Reconfigurable Architectures**

Processors can change their internal structure to adapt to different workloads. This adds a new dimension to TLP, requiring algorithms that dynamically adjust thread execution based on the changing hardware.

References

Otellini, P. (2005, February). *Title of the speech* [Conference session]. Intel Developer Forum, San Francisco, CA, United States.

Che, H., & Nguyen, M. (2014). Amdahl’s law for multithreaded multicore processors. *Journal of Parallel and Distributed Computing, 74*(10), 3056–3069. <https://doi.org/10.1016/j.jpdc.2014.06.012>

Yiapanis, P., Brown, G., & Luján, M. (2016). Compiler-driven software speculation for thread-level parallelism. *ACM Transactions on Programming Languages and Systems, 38*(2), Article 5, 1–45. <https://doi.org/10.1145/2821505>

Gutiérrez, S. K., et al. (2017). Accommodating thread-level heterogeneity in coupled parallel applications. *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 469–478. [invalid URL removed]

Singh, I., Shriraman, A., Fung, W. W. L., O'Connor, M., & Aamodt, T. M. (2013). Cache coherence for GPU architectures. *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, 578–590.

[https://doi.org/10.1109/HPCA.2013.6522351](https://www.google.com/url?sa=E&source=gmail&q=https://doi.org/10.1109/HPCA.2013.6522351)

Gebhart, M., Johnson, D. R., Tarjan, D., Keckler, S. W., Dally, W. J., Lindholm, E., & Skadron, K. (2011). Energy-efficient mechanisms for managing thread context in throughput processors. *SIGARCH Computer Architecture News, 39*(3), 235–246. <https://doi.org/10.1145/2024723.2000093>

"Reconfigurable computing." (2024, June 27). In *Wikipedia*

. Retrieved November 17, 2024, from [https://en.wikipedia.org/w/index.php?title=Reconfigurable\_computing&amp;oldid=1231330546](https://en.wikipedia.org/w/index.php?title=Reconfigurable_computing&oldid=1231330546)

Hennessy, J. L., & Patterson, D. A. (2017). *Computer architecture: A quantitative approach* (6th ed.). Morgan Kaufmann.