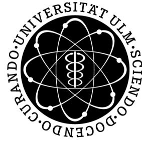

ulm university universität uulm

Fakultät für Ingenieurwissenschaften und Informatik
Institut für Organisation und Management von Informationssystemen

3: Parallelization

Jonas Otto
Matrikelnummer: 982249
ENGG 72323 - Heterogeneous and Parallel Computing Infrastructures
Wordcount: 2.000-3.000

1 Introduction

In recent history, processor performance indicators such as frequency seem to have stagnated. At the same time however, the number of processing cores in a single system has steadily increased (fig. 1). This prompts software developers to adopt a mindset of thinking about parallelization while shaping today's software landscape, to keep up with advances in processor and computer design.

Not all applications are suited to all forms of parallelism, and the scalability and expected performance gain is finite. In this essay, the different ways in which an application can be parallelized, and the factors determining scalability, will be presented and discussed.

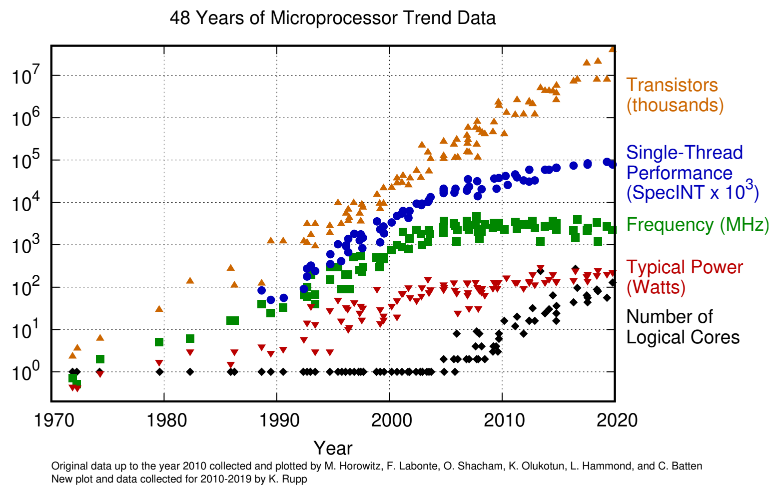

Figure 1: "48 Years of Microprocessor Trend Data" by Karl Rupp, licensed under CC BY 4.0

2 Analysis

2.1 Parallel Computing

A classification of the many parallel computing architectures has been suggested by Flynn. Flynn distinguishes computer architectures by the number of instruction streams (Singe Instruction *SI* or Multiple Instruction *MI*) and number of

data streams (Single Data *SD* or Multiple Data *MD*). Of the four resulting classifications, all but the *MISD* variant are commonly found, and of those the Single Instruction Multiple Data *SIMD* and Multiple Instruction Multiple Data *MIMD* variants are of interest to us here. Most commonly available processors for general purpose computing follow a combination of those approaches: Multicore processors that allow simultaneous execution of multiple programs on separate data (*MIMD* classification) are commonplace. Additionally, the individual processing cores often offer *vector extensions*, which as a form of *SIMD* computing offer fast mathematical operations on multiple values (vectors) at once.

Distributed systems such as entire HPC clusters may also be classified as *MIMD* computing architectures.

It is apparent that *MIMD* architectures in particular are a very far ranging classification, and lend themselves to more granular categorizing. A useful distinction is between shared-memory and non-shared-memory systems. The defining characteristic of a non-shared-memory system is the need for a dedicated communication channel between multiple processing units. On a shared-memory system, this communication can happen implicitly through memory regions accessible to multiple processing units.

On the largest scale, such non-shared-memory, *MIMD* systems such as high performance clusters perform parallel computation by providing a large number of nodes, that each run independently. A fast interconnect between the individual nodes is usually present, but has to be used explicitly (using libraries such as MPI for example).

Single processing nodes (or individual computers, for that matter) are themselves *MIMD* systems, but this time usually of the shared-memory variant. All processors share common RAM, and the operating system provides means of running parallel computation: Processes usually receive their own protected memory segment for safety and security purposes, but multiple threads inside one process can access the same memory without any operating system intervention. The operating system provides scheduling for processes and threads (and has no need to differentiate between a process and thread during scheduling), which allows many programs and programs with more threads than processors to run concurrently. The operating system periodically interrupts execution and selects a different, waiting, thread for execution. Those context switches can yield a substantial performance deficit when many threads are used, and the ratio between program execution time and time spent switching threads is bad.

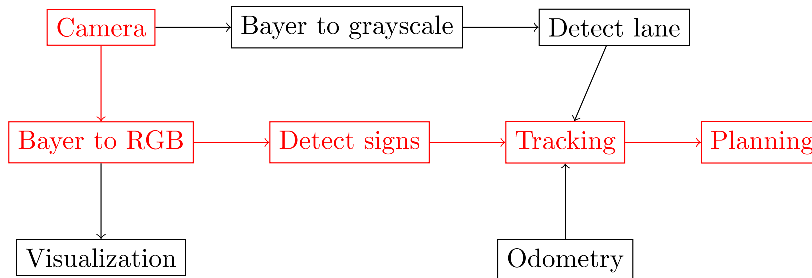

Figure 2: Data flow graph of a fictional task-parallel autonomous-driving application

2.2 Task Parallelism

To benefit from the multiple ways of parallel computing described in section 2.1, an application has to be analyzed in which ways it can be parallelized, and how it scales with an increasing degree of parallelization, and with increasing problem size.

In order to explore and model the ways in which an application can be parallelized, two models are usually considered: Task- and data parallelism.

In this first part, the concept of task parallelism is explored. If an application consists of multiple individual tasks, those can be performed according to its dependencies. Figure 2 illustrates this in terms of a data flow graph: In this example of an autonomous driving application, the individual tasks all require some input data, which is provided by another task. The tracking algorithm for example needs information about both the lane and traffic signs in front of the vehicle. It is however apparent that the tasks of creating a grayscale image and then detecting the lane and the task of creating the RGB image and detecting traffic signs can be executed in parallel. They both depend on the camera image, but execute independently. The serial part of the application continues with the tracking step, which depends on the results of all the previous detection steps. Planning can only be performed after the tracking step.

The *critical path* is the path of execution that determines the execution time of the entire application. In fig. 2, the critical path is marked in red. No matter to which degree the application is parallelized, the total execution time will never be less than the execution time of the critical path, since it's inherently serial. In section 2.4, this will be be called the *serial part* of the application. Another bottleneck arises from the available resources: If, for example, the lane-detection and visualization tasks both require exclusive access to the GPU,

those tasks can not be executed in parallel, even though the data dependencies would allow it, but sequentially or in a switching manner (depending on the scheduling used). There exist frameworks such as “SMP Superscalar” from the Barcelona Supercomputing Center [1] and the OpenCV Graph API that allow the programmer to directly specify the different tasks and dependencies between them. The framework can then execute tasks in parallel and start dependent tasks once their input is ready.

2.3 Data Parallelism

In data-parallel applications, only one single type of task is considered. This task is however applied to a large amount of data, which can be split up to parallelize the application. Common examples of data-parallel algorithms are found in the field of image processing. When applying a filter in the form of convolution with a filter kernel, each pixel in the resulting image could be calculated simultaneously. It only depends on a part of the input image, and the same operation is applied for each pixel.

It has to be ensured that no data dependencies exist between the individual instances of the task, since tasks would have to wait for completion of the previous one otherwise (see section 2.2).

One of the main difficulties in parallelizing a data-parallel application is the segmentation of the data. A segmentation into the smallest possible parts, such as individual pixels of an image, is usually far from optimal. Launching a thread or process always incurs some fixed overhead, and running more threads than processors available increases the overhead of context switching. Other aspects of segmentation have to be kept in mind: Preserving data locality is important to benefit from processor caches, and the particular form of segmentation might help reduce communication, such as by only exchanging data at the border of the segment between iterations. The optimal degree of parallelization can often only be determined empirically.

2.4 Scalability: Amdahl’s Law

Not every part of an application can be parallelized. In section 2.2, it became apparent that tasks often depend on the result of other tasks, which forces them to execute sequentially. In data parallel applications, segmenting the dataset must be done before parallel execution begins.

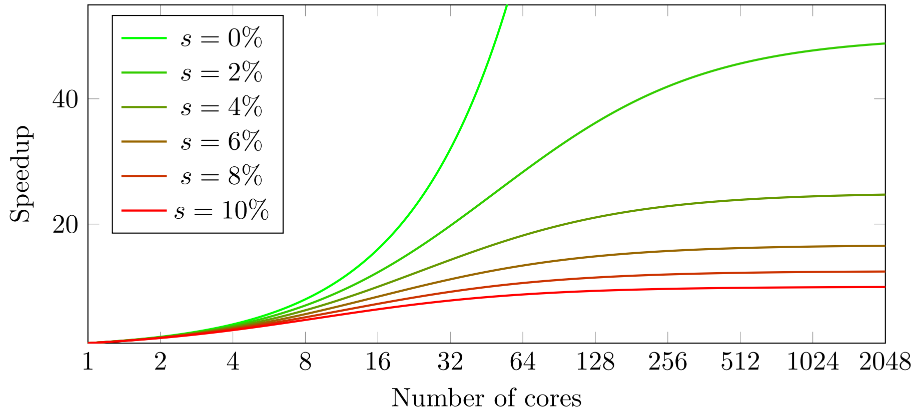

Figure 3: Maximum application with increasing degree of parallelism, according to Amdahl's Law, for varying parts of non-parallelizable code

Named after the computer scientist Gene Amdahl, the maximum possible speedup of an application that consists of a sequential part s and a parallelizable part $p = 1 - s$ can be determined using *Amdahl's Law* [2]: Given n cores or processors, an application which runs in time t before parallelization will have an execution time of

$$t_{\text{parallel}} = s \cdot t + \frac{(1-s) \cdot t}{n} \tag{1}$$

or slower.

Relating the original and parallelized execution time yields the total speedup S:

$$S = \frac{t}{t_{\text{parallel}}} = \frac{n}{n \cdot s + (1-s)} \tag{2}$$

This provides us with a theoretical upper bound for the speedup we can expect when parallelizing an application.

Graphing this relationship of speedup by number of cores, for varying amounts of serial part s, results in fig. 3. It is apparent that even for applications that are largely parallelizable, the benefit of adding additional processing units diminishes quickly. Figure 4 shows this relationship for fixed n, and it is apparent that significant performance gains require a sufficiently large parallelizable part.

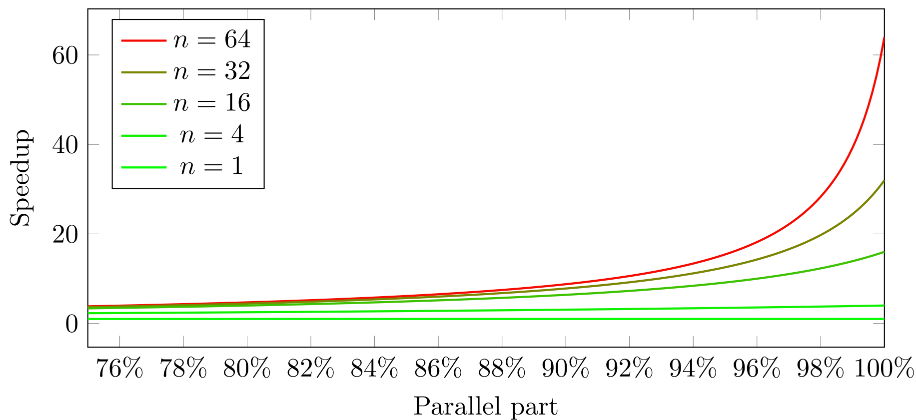

Figure 4: Amdahl's law graphed for fixed number of processing units n

2.5 Limitations of Amdahl's Law

Amdahl's law provides an upper bound, but not one that a programmer can reasonably expect to immediately reach. Even with trivially parallel problems, the overhead induced by parallelizing is never zero. Communication, organization and management of parallel execution threads takes time, and increases with the number of threads. We expect to reach a point where the communication overhead dominates the execution time compared to the actual application task. The speedup decreases, and may even become negative, meaning the extra overhead makes the parallel application run slower than the fully serial one. In section 2.1, it was already mentioned that an operating system, which interrupts the running thread to run more threads than available processors, can be devastating to performance. While the impact of the operating system will not be discussed in detail here, this is important to consider when designing parallel programs.

3 Discussion

In the following, the parallelization of applications shall be considered from an implementation standpoint. How do the specific ways in which an application exploits parallel computation differ? How does that impact the software design and architecture? And in which layers of abstraction can the parallelization be hidden?

3.1 Parallelism at application level

Direct, explicit parallelization of an application might be the most straightforward way of running the program on a parallel architecture. This does however mean that several challenges have to be addressed: Threads or processes have to be started, the problem has to be partitioned, work has to be distributed. Communication has to be established manually, and synchronization between threads is necessary.

On this level, operating-system functions such as POSIX `pthreads` are used for creating and managing threads. Synchronization primitives such as semaphores or language specific implementations like the C++ `std::mutex` are used to manage access to shared resources. Communication between threads can be performed implicitly, by writing results into a shared memory segment or by returning results directly.

A popular pattern for parallel applications is the *fork-and-join* pattern: The program repeatedly forks, spawning multiple threads that each work on a portion of the current task, and then *joins* those threads, waiting for each to finish execution and thus providing a point of synchronization.

The *master-worker* pattern describes a situation in which the work is continuously packaged into tasks, which the *master* then submits for completion by a *worker*. Both of these popular design patterns can be combined, for example by submitting tasks to an existing thread pool instead of actually `fork`ing the process or creating new threads, in order to avoid some of the overhead induced by thread-creation.

An application programmer is however not required to use `pthreads` and friends manually. Libraries such as Intel TBB and compiler extensions like OpenACC [3] and OpenMP [4] exist and provide developers with facilities like thread pools. OpenMP even offers automatic parallelization of `for` loops for example, which implicitly divides the loop into tasks submitted to the internal thread pool, and takes care of joining threads and collecting results such that the program continues as if execution was sequential.

3.2 Parallelism below the application level

In the interest of hiding implementation details from high-level programs, parallelism can be hidden from the application programmer: If a software library offers a sufficiently high level of abstraction, it can perform internal operations

in parallel while maintaining a serial programming model to the developer. One example is the popular image processing library OpenCV: OpenCV hides complex algorithms behind a relatively simple interface. Many of those algorithms in the field of image processing lend themselves nicely to a data-parallel approach and benefit greatly from parallelization. Therefore, OpenCV maintains a thread pool to execute those operations, and even contains GPU implementations using OpenCL and CUDA for some algorithms.

A similar approach is even taken by ubiquitous libraries such as the C++ *Standard Template Library (STL)*: Since 2017, many of the functions in the `<algorithm>` library take an additional *execution policy* argument that allows the programmer to specify that the algorithm shall be parallelized, vectorized, both, or neither. The concrete implementation varies by library vendor, but an internally maintained thread pool is used at least by GCC‘s `libstdc++` (using Intel TBB internally [5]).

3.3 Parallelism above the application level

A different approach to parallelizing application code which is not written in an inherently parallel way is to exploit the fact that the application may already be divided into more or less independent modules, which can be executed in parallel. This is often the case, when the application code is embedded into some kind of framework. The Robot Operating System ROS for example is a framework popular for application in the fields of robotics and automation. A core concept of this framework is the notion of a node, which is a program that receives and publishes data via publish/subscribe channels and performs a specific task (such as receiving sensor data or controlling actuators). Those nodes can be started in individual processes (or threads), since they only rely on the publish/subscribe communication channels for synchronization.

While this can lead to an immediate performance increase compared to serial execution, this does not provide scalability with more processors. The upper bound of performance increase is reached as soon as every node has enough processing resource to not have to share them with another node (disregarding the potential of each single node to benefit from multiple processors).

A similar effect can be achieved using MPI, which also provides an inter-process communication channel and starts multiple processes, although in this case the individual processes usually perform the same task on a smaller subset of data, which enables greater scalability with the number of processes. This could be

considered an example for data parallelism, while the ROS example is closer to task parallelism.

4 Conclusion

Parallel processing architectures enable great advances in computational performance despite stagnating clock speeds. To benefit from this development, application programmers have to be aware of how execution time scales with an ever increasing degree of parallelism, how to exploit the different ways a computer architecture realizes parallel execution, and where the pitfalls lie when programming a parallel system.

Amdahl's law illustrates how the effect of additional processing elements diminishes the higher the inherently serial share of an application is. It determines where the theoretical limits are in terms of expected speedup for a given application with a varying degree of parallelism.

Task- and Data-Parallelism are two models that can be used to describe the parallel nature of a problem, and help to transform that problem into a parallel application. Finally, multiple approaches for specific implementations have been discussed, from the viewpoint of a layered application utilizing frameworks and libraries. There is no best way to parallelize any application, and no one recipe for reaching the theoretical peak performance. But if the developer is conscious about the inherently serial part of the application, and the implications thereof, tools, libraries and frameworks exist to take advantage of parallel computing architectures at any point in the development process.

References

[1] L. Schubert, "Heterogeneous and Parallel Computing Infrastructures," University of Ulm, 2020.

[2] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, ser. AFIPS '67 (Spring). New York, NY, USA: Association for Computing Machinery, 1967, p. 483–485. [Online]. Available: https://doi.org/10.1145/1465482.1465560

[3] OpenACC-Standard.org, "The OpenACC application programming interface," 2020. [Online]. Available: https://www.openacc.org/specification

[4] OpenMP Architecture Review Board, "OpenMP application program interface version 3.0," 2020. [Online]. Available: https://www.openmp.org/spec-html/5.1/openmp.html

[5] The Free Software Foundation and contributors, "The GNU C++ Library Manual," 2020. [Online]. Available: https://gcc.gnu.org/onlinedocs/libstdc++/manual/index.html

Declaration of Originality

I confirm that this assignment is my own work and that I have not sought or used inadmissible help of third parties to produce this work and that I have clearly referenced all sources used in the work. I have fully referenced and used inverted commas for all text directly or indirectly quoted from a source.

This work has not yet been submitted to another examination institution – neither in Germany nor outside Germany – neither in the same nor in a similar way and has not yet been published.

Ulm, on the ....

Jonas Otto