KARLSRUHE INSTITUTE OF TECHNOLOGY

Mehrdimensionale Signalverarbeitung und Bildauswertung mit Graphikkarten und anderen Mehrkernprozessoren

Manuel Lang

March 30, 2018

Contents

1 Introduction

- Parallel computing: Use of multiple (interacting) computational units(CU) to execute a (divisible) task.
- Amdahls law: We want to know how fast we can complete a task on a particular data set by increasing the CU count. $\eta_n = \frac{TW}{T(n)W} = \frac{T}{t_s + \frac{t_p}{n} + t_{comm}}$
  - η_n: Speeup
  - W: Work load
  - T: Total runtime $t_s + t_p$
  - t_s: Runtime serial part
  - t_p: Runtime parallel part
  - t_{comm}: Communication time (normally not included)
  - n: Number of computational units
- Gustafson's law: We want to know if we can analyze more data in approximately the same amount of time by increasing the CU count. $\eta_n = \frac{TW(n)}{TW} = (1-p)W + npW$
  - η_n: Speeup
  - W: Work load
  - T: Total runtime
  - p: Workload fraction benefiting from additional CUs
  - $W(n) : (1-p)W + npW \equiv aW + n(1-a)W = nW - a(n-1)W$
  - n: Number of computational units
- Data parallelism: Each CU performs the same task on dierent data
  - The CPU cores and GPU streaming-cores are OpenCL compute devices
  - Concurrent processing on all herterogeneous cores.
- Task parallelism: Each CU performs a different task on the same data
- Instruction level parallelism: Automatic parallel execution of instructions by processor
- Spatial parallelism: More units work in parallel
- Temporal parallelism → Pipelining
- Latency
  - The latency of an instruction is the **delay** that the instruction generates in a dependy chain. The measurement unit is clock cycles.
  - CPUs try to minimize latency. Low efficiency on parallel portions.

- Throughput
  - The throughput is the maximum number of instructions of the same kind that can be executed per clock cycle when the operands of each instruction are independant of the preceding instrucions.
  - GPUs try to maximize throughput. Low performance on sequentiel portions.
- Graphic card slang
  - A GPU executes a program, the *kernel*.
  - A thread executes an instance of the kernel.
  - Threads are combined into warps/wavefronts running in lockstep. Individual threads composing a warp start together at the same program address, but they have their own instruction address counter and register state and are therefore free to branch and execute independently.
  - Warps/wavefronts are part of threaded blocks/work groups. These are defined by the user.
  - A thread runs on a core. A number of cores form a *Streaming Multiprocessor (SM)* / *Compute Unit (CU)*. Thread blocks/work groups are scheduled over SMs/CUs.

2 Parallelism / Programming models

- Foster's PCAM model
  - Tasks communicate over channels
    - Task: A program with local memory, in- and outports. Tasks execute concurrently. The number of tasks can vary during program execution.
    - Channel: Connecting outports of a task with import of another task. Channels are buffered sending asynchronously and receiving synchronously (task blocks).
  - Paritioning: The computation that is to be performed and the data operated on by this computation are decomposed into small tasks. Practical issues such as the number of processors in the target computer are ignored, and attention is focused on recognizing opportunities for parallel execution.
    - The partitioning stage of a design is intended to expose opportunities for parallel execution.
    - Focus is on defining a large number of small tasks (fine-grained decomposition).
    - A good partition divides both the computation and the data into small pieces.
    - One approach is to focus first on partitioning the data associated with a problem; this is called domain decomposition.

        - First partition data; ideally divide data into small pieces of approximately equal size.
        - Next partition computation, typically by associating each operation with the data on which it operates.
        - Focus first on the largest data structure or on the data structure that is accessed most frequently.
    * The alternative approach, termed functional decomposition, decomposes the computation into seperate tasks before considering how to partition the data.
        - Initital focus is on the computation that is to be performed rather than on the data.
        - Divide computation into disjoint tasks.
        - Examine data requirements of tasks:
            1. Requirements may be disjoint, in which case the partition is complete.
            2. Requirements may overlap significantly, in which case considerable communication will be required to avoid replication of data. Domain decomposition should be considered instead.
        - Functional decomposition is valuable as a different way of thinking about problems and should be considered when exploring possible parallel algorithms.
        - A focus on the computations that are to be performed can sometimes reveal structure in a problem, and hence opportunities for optimization, that would not be obvious from a study of data alone.
        - Functional decomposition is an important program structuring technique; can reduce the complexity of the overall design.
    * Theses are complementary techniques.
    * Seek to avoid replicating computation and data (may change this later in process).
    * Paritioning checklist
        - Does your partition define at least an order of magnitude more tasks than there are processors in your target computer?
        - Does your partition avoid redundant computation and storage requirements?
        - Are tasks of comparable size?
        - Have you identified several alternative partitions?
- Communication: The communication required to coordinate task execution is determined, and appropriate communication structures and algorithms are defined.

    - Conceptualize a need for communication between two tasks as a channel linking the tasks, on which one task can send messages and from which the other can receive.
    - Channel structure links tasks that require data (consumers) with tasks that possess those data (producers).
    - Definition of a channel involves an intellectual cost and the sending of a message involves a physical cost - avoid introducing unnecessary channels and communication operations.
    - We want to distribute communication operations over many tasks.
    - We want to organize communication operations in a way that permits concurrent execution.
    - For functional decomposition communication is often 'natural'.
    - For domain decomposition:
        - First partition data structures into disjoint subsets and then associate with each datum those operations that operate solely on that datum.
        - Handle dependencies with other tasks.
    - Communication patterns
        - local vs. global
        - structured vs. unstructured
        - static vs. dynamic
        - synchronous vs. asynchronous
    - Communication checklist
        - Do all tasks perform about the same number of communication operations?
        - Does each task communicate only with a small number of neighbors?
        - Are communication operations able to proceed concurrently?
        - Is the computation associated with different tasks able to proceed concurrently?
- Agglomeration: The task and communication structures defined in the first two stages of a design are evaluated with respect to performance requirements and implementation costs. If necessary, tasks are combined into larger tasks to improve performance or to reduce development costs.
    - Move from parallel abstractions to real implementation. At this point we've broken down our problem enough that we understand the individual tasks and the necessary communication between tasks. The goal now is to being making the parallel solution practical and as efficient as possible.

    - Is it useful to combine, or agglomerate, tasks to reduce the number of tasks?
    - Is it worthwile to replicate data and/or computation?
* Agglomeration goals
    - Reducing communication costs by increasing computation and communication granularity.
    - Retaining flexibility with respect to scalability and mapping decisions.
* Surface-to-Volume effects
    - The communication requirements of a task are proportional to the surface of the subdomain on which it operates, while the computation requirements are proportional to the subdomain's volume. In a two-dimensional problem, the surface scales with the problem size while the volume scales as the problem size squared.
    - The amount of communication performed for a unit of computation (the communication/computation ratio) decreases as task size increases. This effect is often visible when a partition is obtained by using domain decomposition techniques.
* Replication of computations: Sometimes it's more efficient for a task to compute a needed quantity rather than to receive it from another task where it is already known or has been computed.
* Avoid communication: Agglomeration is almost always beneficial if analysis of communication requirements reveals that a set of tasks cannot execute concurrently.
* Preserving flexibility
    - It is important when agglomerating to avoid making design decisions that limit unnecessarily an algorithm's scalability. For example, we might choose to decompose a multidimensional data structure in just a single dimension.
    - Don't assume during the design that the number of processors will always be limited to the currently available number.
    - Good parallel algorithms are designed to be resilient to changes in processor count.
    - It can be advantageous to map several tasks to a processor. Then, a blocked task need not result in a processor becoming idle, since another task may be able to execute in its place.
* Agglomeration checklist
    - Has agglomeration reduced communication costs by increasing locality?

- If agglomeation has replicated communication, have you verified that the benefits of this replication outweigh its costs, for a range of problem sizes and processor counts?
- If agglomeration replicates data, have you verified that this does not compromise the scalability of your algorithm by restricting the range of problem sizes or processor counts that it can address?
- Has agglomeration yielded tasks with similar computation and communication costs? The larger the tasks created by agglomeration, the more important it is that they have similar costs.
- Does the number of tasks still scale with problem size?
- If agglomeration eliminated opportunities for concurrent execution, have you verified that there is sufficient concurrency for current and future target computers?
- Can the number of tasks be reduced still further, without introducing load imbalances or reducing scalability?

– Mapping: Each task is assigned to a processor in a manner that attempts to satisfy the competing goals of maximizing processor utilization and minimizing communication costs. Mapping can be specified statically or determined at runtime by load-balancing algorithms.
  * At this point we have a set of tasks and we need to assign them to processors on the available machine. The mapping problem does not arise on uniprocessors or on shared-memory computers that provide automatic task scheduling.
  * General-purpose mapping mechanisms have yet to be developed for scalable parallel computers. The general-case mapping problem is NP-complete.
  * Our goal in developing mapping algorithms is normally to minimize total execution time. We use two strategies to achieve this goal:
    1. We place tasks that are able to execute concurrently on different processors, so as to enhance concurrency.
    2. We place tasks that communicate frequently on the same processor, so as to increase locality.
  * Considerable knowledge has been gained on specialized strategies and heuristics and the classes of problem for which they are effective.
  * When domain decomposition is used there is often a fixed number of equal-size tasks and structured local and global communication.
  * If, instead, there are variable amounts of work per task and/or unstructured communication patterns, we might use load balancing algorithms that seek to identify efficient agglomeration and mapping strategies. The time required to execute these algorithms must be weighed against the benefits of reduced

execution time. Probabilistic load-balancing methods tend to have lower overhead than do methods that exploit structure in an application.

    * When either the number of tasks or the amount of computation or communication per task changes dynamically during program execution we might use dynamic load-balancing strategy in which a load-balancing algorithm is executed periodically to determine a new agglomeration and mapping.
    * If functional decomposition is used we can use task-scheduling algortihms which allocate tasks to processors that idle or that are likely to become idle.

- Parallel Patterns
  - The Berkeley View
    * Seven critical questions for 21st century parallel computing
      · Applications
      · What are the applications?
      · What are common kernels of the applications?
      · Hardware
      · What are the hardware building blocks?
      · How to connect them?
      · Programming models
      · How to describe applications and kernels?
      · How to program the hardware?
      · Evaluation
      · How to measure success?
    * 13 Dwarfs
      1. Finite State Machine
      2. Combinational Logic
      3. Graph Traversal
      4. Structured Grids
      5. Dense Linear Algebra
      6. Sparse Linear Algebra
      7. Spectral Methods (FFT)
      8. Dynamic Programming
      9. N-Body Methods
      10. MapReduce
      11. Back-track/Branch & Bound

12. Graphical Model Inference
13. Unstructured Grids

* 4 Valuable Roles of Dwarfs
    1. "Anti-benchmarks" not tied to code or language artifcats $\Longrightarrow$ encourage innovation in algorithms, languages, data structures, and/or hardware
    2. Universal, understandable vocabulary to talk across disciplinary boundaries
    3. Define building blocks for creating libraries & frameworks that cut across app domains
    4. They decouplse research, allowing analysis of HW & SW engineering and programming without waiting years for full apps
* Parallel Patterns
    - A recurring combination of tasks distribution and data access that solves a specific problem in parallel algorithm design.
    - Patterns can be used to organize your code, leading to algorithms that are most scalable and maintainable.
    - Patterns are universal - they can be used in any parallel programming system.
    - Good parallel programming models support a set of useful parallel patterns with low-overhead implementations.
* Serial Control Patterns
    - Sequence: A sequence is a ordered list of tasks that are executed in a specific order. Each task is completed before the one after it starts.
    - Selection: In the selection pattern, a condition c is evaluated first. If the condition is true, some task a is executed. If c is false, task b will be executed. There is a control-flow dependency between c and a,b, so neither a or b is executed before c. Exactly one of a or b will be executed, never both.
    - Iteration: In the iteration pattern, a condition c is evaluated. If it is true, a task a is evaluated, then c is evaluated again, and the process repeats until the condition becomes false. The number of iterations is data dependend.
    - Recursion: Recursion is a dynamic form of nesting which allows functions to call themselves, directly or indirectly.
* Parallel Control Patterns
    - Fork-Join: The fork-join pattern lets control flow fork into multiple parallel flows that rejoin later.

- Map: The map pattern replicates a (elemental) function over every element of an index set. The index set may be abstract or associated with the elements of a acollection.
- Stencil: Stencil applies a function to neighbourhoods of a collection.
- Reduction: Combines every element in a collection using an associative "combiner function". Because of the associativity of the combiner function, different orderings of the reduction are possible.
- Scan: Computes all partial reduction of a collection. For every output in a collection, a reduction of the input up to that point is computed. If the function being used is associative, the scan can be parallelized. Parallelizing a scan is not obvious at first, because of dependencies to previous iterations in the serial loop. A parallel scan will require more operations than a serial version.
- Recurrence: Recurrence results from loop nests with both input and output dependencies between iterations. For a recurrence to be computable, there must be a serial ordering of the recurrence elements so that elements can be computed using previously computed outputs.

* Serial Data Management Patterns
  - Random Read and Write
  - Stack Allocation
  - Heap Allocation
  - Closures
  - Objects
* Parallel Data Management Patterns
  - Pack: Pack is used eliminate unused space in a collection. Elements marked false are discarded, the remaining elements are placed in a contiguous sequence in the same order. Useful when used with map.
  - Pipeline: A Pipeline connects tasks in a producer-consumer manner. Some stages may retain state. Pipelines are most useful when used with other patterns as they can multiply available parallelism.
  - Geometric Decomposition: The geometric decomposition arranges data into subcollections. Overlapping and non-overlapping decompositions are possible.
  - Gather: Gather reads a collection of data given a collection of indices. The output collection shares the same type as the input collection, but it share the same shape as the indices collection.
  - Scatter: Scatter is the inverse of gather. A set of input and indices is required, but each element of the input is written to the output at the given

index instead of read from the input at the given index. Race conditions can occur when we have two writes to the same location.

    * Other parallel patterns
        · Workpile: general map pattern where each instance of elemental function can generate more instances, adding to the "pile" of work.
        · Search: finds some data in a collection that meets some criteria.
        · Segmentation: operations on subdivided, nonoverlapping, non-uniformly sized partitions of 1D collections.
        · Expand: a combination of pack and map. Each map can output any number of elements. The outputs are packed in a specific order.
        · Category Reduction: Given a collection of elements each with a label, find all elements with same label and reduce them.
- The Intel View