Parallel Programming Concepts

Parallel Algorithms

Peter Tröger

Sources:

Ian Foster. Designing and Building Parallel Programs. Addison-Wesley. 1995.
Kurt Keutzer (EECS UC Berkeley) and Tim Mattson (Intel) - A Design Pattern Language for Engineering (Parallel) Software
Why Parallel?

- P is the portion of the program that benefits from parallelization

- Amdahl’s Law (1967)
  - Maximum speedup $s_{Amdahl}$ by $N$ processors
    
    $$s_{Amdahl} = \frac{(1-P)+P}{(1-P)+\frac{P}{N}}$$
  
  - Largest impact of parallelization with small $N$ and/or small $(1-P)$
  
  - Speedup by increasing $N$ is limited

- Gustafson’s Law (1988)
  
  - Maximum speedup $s_{Gustafson}$ by $N$ processors
    
    $$s_{Gustafson} = \frac{(1-P)_N+N*P_N}{(1-P)_N+P_N}$$
    
    $$= (1 - P)_N + N * P_N$$
  
  - Assumption: Problem size grows with $N$, so the inherently serial portion becomes smaller as proportion to the overall problem
  
  - With neglection of the parallelization overhead, speedup can grow as $N$
Amdahl's Law

The graph illustrates the relationship between the number of processors and the speedup according to Amdahl's Law. The speedup is shown on the y-axis, while the x-axis represents the number of processors. The graph includes curves for different percentages of parallelizable parts (P): 95%, 90%, 75%, 50%, 25%, and 10%. The speedup decreases as the number of processors increases, especially for lower percentages of parallelizable parts. The concept of Amdahl's Law is crucial in understanding the limitations of parallel processing and how increasing the number of processors does not always lead to proportionally increased performance.
Why Parallel?

  - Measure degree of code parallelization, by determining serial fraction through experimentation
  - Rearrange Amdahl’s law for sequential portion
  - Allows computation of empirical sequential portion, based on measurements of execution time, without code inspection

\[
S = \frac{Speed_N - \frac{1}{N}}{1 - \frac{1}{N}}
\]

\[
Speed_N = S + \frac{P}{N} = S + \frac{1-S}{N}
\]
Parallel Algorithms and Design Patterns

• Vast body of knowledge in books and scientific publications

• Typically discussion based on abstract machine model (e.g. PRAM), to allow theoretical complexity analysis

• Rule of thumb: Somebody else is smarter than you - reuse !!


  • Herlihy, Maurice; Shavit, Nir: The Art of Multiprocessor Programming. Morgan Kaufmann, 2008. , 978-0123705914

  • ParaPLoP - Workshop on Parallel Programming Patterns

  • ’Our Pattern Language‘ (http://parlab.eecs.berkeley.edu/wiki/patterns/)

  • Programming language support libraries
Designing Parallel Algorithms [Breshears]

• Parallel solution must keep *sequential consistency* property

• „Mentally simulate“ the execution of parallel streams on suspected parts of the sequential application

• Amount of computation per parallel task must offset the overhead that is always introduced by moving from serial to parallel code

• *Granularity*: Amount of computation done before synchronization is needed
  
  • **Fine-grained granularity** overhead vs. **coarse-grained granularity** concurrency

  • Iterative approach of finding the right granularity

  • Decision might be only correct only for the execution host under test

• Execution order dependency vs. data dependency
Designing Parallel Algorithms [Foster]

- Translate problem specification into an algorithm achieving concurrency, scalability, and locality

- Best parallel solution typically differs massively from the sequential version

- Four distinct stages of a methodological approach
  
  - Search for concurrency and scalability:
    - 1) **Partitioning** - decompose computation and data into small tasks
    - 2) **Communication** - define necessary coordination of task execution

  - Search for locality and other performance-related issues:
    - 3) **Agglomeration** - consider performance and implementation costs
    - 4) **Mapping** - maximize processor utilization, minimize communication

- Might require backtracking or parallel investigation of steps
Partitioning Step

• Expose opportunities for parallel execution - fine-grained decomposition

• Good partition keeps computation and data together
  • Starting with data partitioning leads to domain / data decomposition
  • Computation partitioning leads to functional / task decomposition
  • Complementary approaches, can lead to different algorithm versions
  • Reveal hidden structures of the algorithm that have potential -> investigate complementary views on the problem

• Avoid replication of either computation or data, can be revised later to reduce communication overhead

• Step results in multiple candidate solutions
Partitioning - Decomposition Types

• Domain Decomposition
  • Define small data fragments, then specify computation for them
  • Different phases of computation on the same data are handled separately
  • Rule of thumb: First focus on large or frequently used data structures

• Functional Decomposition
  • Split up computation into disjoint tasks, ignore the data accessed for the moment
  • Example: Producer / consumer
  • With significant data overlap, domain decomposition is more appropriate
Partitioning Strategies [Breshears]

• Loop parallelization
  • Reason about code behavior when loop would be executed backwards - strong indicator for independent iterations

• Produce at least as many tasks as there will be threads / cores
  • But: Might be more effective to use only fraction of the cores (granularity)
  • Computation part must pay-off with respect to parallelization overhead

• Avoid synchronization, since it adds up as overhead to serial execution time

• Patterns for data decomposition: by element, by row, by column group, by block
  • Influenced by surface-to-volume ratio
Partitioning - Checklist

• Checklist for resulting partitioning scheme
  • Order of magnitude more tasks than processors?
    -> Keeps flexibility for next steps
  • Avoidance of redundant computation and storage requirements?
    -> Scalability for large problem sizes
  • Tasks of comparable size?
    -> Goal to allocate equal work to processors
  • Does number of tasks scale with the problem size?
    -> Algorithm should be able to solve larger tasks with more processors
• Resolve bad partitioning by estimating performance behavior, and eventually reformulating the problem
Communication Step

• Specify links between data consumers and data producers

• Specify kind and number of messages on these links

• Domain decomposition problems might have tricky communication infrastructures, due to data dependencies

• Communication in functional decomposition problems can easily be modeled from the data flow between the tasks

• Categorization of communication patterns
  • *Local* communication (few neighbors) vs. *global* communication
  • *Structured* communication (e.g. tree) vs. *unstructured* communication
  • *Static* vs. *dynamic* communication structure
  • *Synchronous* vs. *asynchronous* communication
Communication - Hints

- Distribute computation and communication, don’t centralize algorithm
  - Bad example: Central manager for parallel reduction
  - *Divide-and-conquer* helps as mental model to identify concurrency
- Unstructured communication is hard to agglomerate, better avoid it

- Checklist for communication design
  - Do all tasks perform the same amount of communication?
    - -> Distribute or replicate communication hot spots
  - Does each task performs only local communication?
  - Can communication happen concurrently?
  - Can computation happen concurrently?
Ghost Cells

• Domain decomposition might lead to chunks that demand data from each other for their computation

  • Solution 1: Copy necessary portion of data (‘ghost cells‘)
    • Feasible if no synchronization is needed after update
    • Data amount and frequency of update influences resulting overhead and efficiency
    • Additional memory consumption

  • Solution 2: Access relevant data ‘remotely‘ as needed
    • Delays thread coordination until the data is really needed
    • Correctness („old“ data vs. „new“ data) must be considered on parallel progress
Agglomeration Step

- Algorithm so far is correct, but not specialized for some execution environment
- Check again partitioning and communication decisions
  - Agglomerate tasks for more efficient execution on some machine
  - Replicate data and / or computation for efficiency reasons
- Resulting number of tasks can still be greater than the number of processors
- Three conflicting guiding decisions
  - Reduce communication costs by *coarser granularity* of computation and communication
  - *Preserve flexibility* with respect to later mapping decisions
  - Reduce software engineering costs (serial -> parallel version)
Agglomeration [Foster]
Agglomeration - Granularity vs. Flexibility

- Reduce communication costs by coarser granularity
  - Sending less data
  - Sending fewer messages (per-message initialization costs)
  - Agglomerate tasks, especially if they cannot run concurrently anyway
    - Reduces also task creation costs
  - Replicate computation to avoid communication (helps also with reliability)
- Preserve flexibility
  - Flexible large number of tasks still prerequisite for scalability
- Define granularity as compile-time or run-time parameter
Agglomeration - Checklist

• Communication costs reduced by increasing locality ?
• Does replicated computation outweighs its costs in all cases ?
• Does data replication restrict the range of problem sizes / processor counts ?
• Does the larger tasks still have similar computation / communication costs ?
• Does the larger tasks still act with sufficient concurrency ?
• Does the number of tasks still scale with the problem size ?
• How much can the task count decrease, without disturbing load balancing, scalability, or engineering costs ?
• Is the transition to parallel code worth the engineering costs ?
Mapping Step

- Only relevant for distributed systems, since shared memory systems typically perform automatic task scheduling

- Minimize execution time by
  - Place concurrent tasks on different nodes
  - Place tasks with heavy communication on the same node

- Conflicting strategies, additionally restricted by resource limits
  - In general, NP-complete bin packing problem

- Set of sophisticated (dynamic) heuristics for load balancing
  - Preference for local algorithms that do not need global scheduling state
Surface-To-Volume Effect [Foster, Breshears]

- **Communication** requirements of a task are proportional to the **surface** of the data part it operates upon - amount of 'borders' on the data

- **Computational** requirements of a task are proportional to the **volume** of the data part it operates upon - granularity of decomposition

- **Communication / computation ratio** decreases for increasing data size per task

- Better to have coarse granularity by agglomerating tasks in all dimensions
  - For given volume (computation), the surface area (communication) then goes down -> good
Surface-to-Volume Effect [Foster]

- Computation on 8x8 grid
  - (a): 64 tasks, one point each
    - 64x4 = 256 communications
    - 256 data values are transferred
  - (b): 4 tasks, 16 points each
    - 4x4 = 16 communications
    - 16x4 = 64 data values are transferred
Patterns for Parallel Programming [Mattson]

- Categorization of general parallelization concepts as linear hierarchy

  - **Finding Concurrency Design Space** - task / data decomposition, task grouping and ordering due to data flow dependencies, design evaluation
    - Identify and analyze exploitable concurrency
  
  - **Algorithm Structure Design Space** - task parallelism, divide and conquer, geometric decomposition, recursive data, pipeline, event-based coordination
    - Mapping of concurrent design elements to units of execution
  
  - **Supporting Structures Design Space** - SPMD, master / worker, loop parallelism, fork / join, shared data, shared queue, distributed array
    - Program structures and data structures used for code creation
  
  - **Implementation Mechanisms Design Space** - threads, processes, synchronization, communication
Data Decomposition [Mattson]

- Good strategy if ...
  - ... most computation is organized around the manipulation of a large data structure
  - ... similar operations are applied to different parts of the data structure
- Data decomposition is often driven by needs from task decomposition
- Array-based computation (row, column, block), recursive structures
- In a good data decomposition, dependencies scale at lower dimension than the computational effort for each chunk
- Example: Matrix multiplication
  - C=A*B - decompose C into row blocks, requires full B, but only the corresponding A row block

(C) Wikipedia
Task Grouping [Mattson]

- Consider constraints for task groups, not for single items
  - Temporal dependency - Data flow from group A to group B necessary
  - Semantics - Group members have to run at the same time (fork / join)
  - Independent task groups - Clear identification for better scheduling
- Finding task groups, based on abstract constraints
  - Tasks that correspond to a high-level operation naturally group together
  - If tasks share a constraint (e.g. data), keep them as distinct group
  - Merge groups with same constraints
Data Sharing [Mattson]

- In addition to task-local data, central dependency to shared data exists
  - Tasks might also need other tasks data, global shared read does not scale
- Analyze shared data according to its class
  - *Read-Only*: no protection overhead necessary
  - *Effectively-local*: data partitioned into independent sub sets, no locking
  - *Read-write*: global behavior must comply to a consistency model
    - *Accumulate*: Each task has local copy, final accumulation to one result
    - *Multiple-read / single-write*: Data decomposition problems
- Define abstract type with according operations
- Solve by one-time-execution, non-interfering operations, reader / writer
Algorithm Design Evaluation [Mattson]

- Minimal consideration of suitability for target platform
  - Number of processing elements and data sharing amongst them
  - System implications on physical vs. logical cores
  - Overhead for technical realization of dependency management (e.g. MPI)

- Flexibility criteria
  - Flexible number of decomposed tasks supported?
  - Task definition independent from scheduling strategy?
  - Can size and number of chunks be parameterized?
  - Are boundary cases handled correctly?
Algorithm Structure Design Space [Mattson]

- Organize by tasks
  - Linear -> Task Parallelism
  - Recursive -> Divide and Conquer (e.g. Merge Sort)
- Organize by Data Decomposition
  - Linear -> Geometric decomposition
  - Recursive -> Recursive Data
- Organize by Flow of Data
  - Regular -> Pipeline
  - Irregular -> Event-Based Coordination
Supporting Structures [Mattson]

- Program structures
  - Single-program-multiple-data (SPMB)
  - Master / worker
  - Loop parallelism
  - Fork / Join

- Data structures
  - Shared data
  - Shared queue
  - Distributed array
What‘s Not Parallel [Breshears]

• Algorithms with state that cannot be handled through parallel tasks (e.g. I/O)
• Recurrence relations - each loop run is a function of the previous one
  • Example: Fibonacci
• Reduction - take arrays of values and reduce them to a single value
  • For associative and commutative operators, parallelization is possible
• Loop-carried dependency - use results of previous iterations in loop body

```c
for (n=0; n<=N; ++n) {
    opt[n] = Sn;
    Sn *= 1.1; }
```