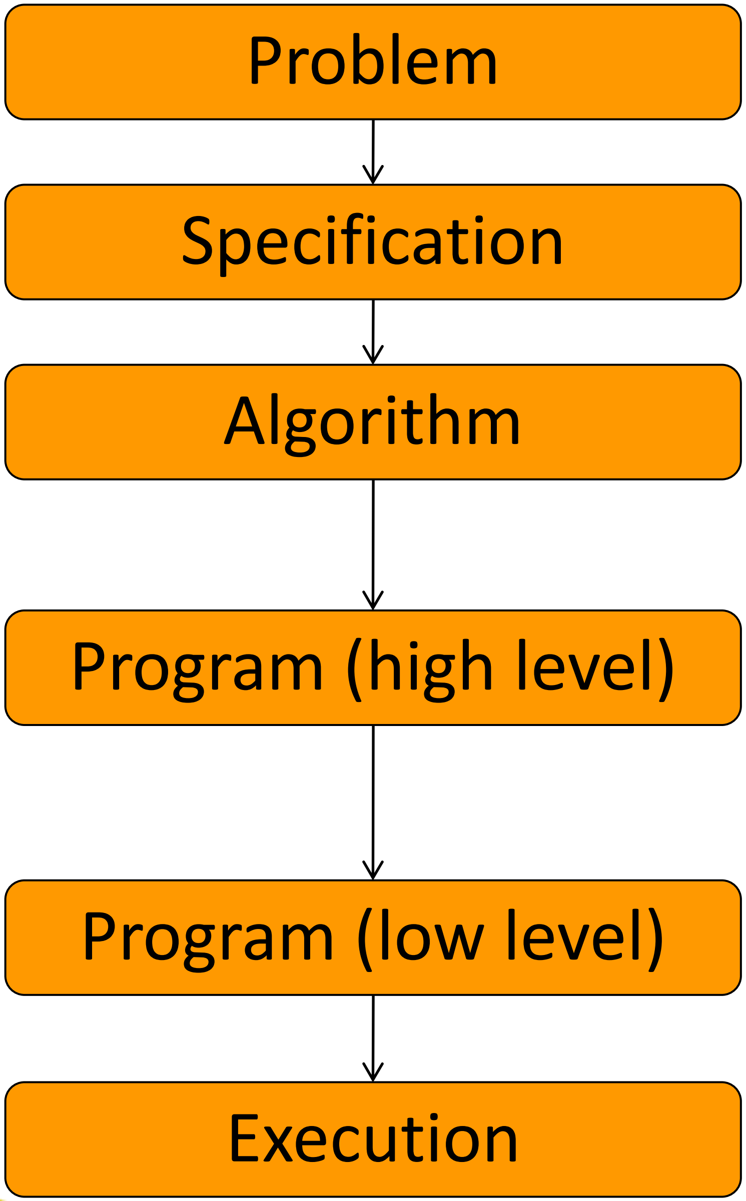
Parallel Programming

Parallel Algorithm Design

Jan Schönherr
Technische Universität Berlin
School IV – Electrical Engineering and Computer Sciences
Communication and Operating Systems (KBS)
Einsteinufer 17, Sekr. EN6, 10587 Berlin

Parallelism in Software Development



Deliberately free of implementation aspects

Identify potential parallelism; design parallel algorithms; make use of TLP and DLP

Make use of a library enabling parallelism or use parallel constructs of a parallel or augmented sequential language

Compiler prepares for ILP, converts parallel constructs into DLP+TLP. Might add DLP+TLP.

CPU realizes ILP

Foster's Design Methodology

- Encourages scalable parallel algorithms
  - Targets mainly distributed memory system
  - Applicable to other systems as well
- Published 1995

- Four steps
  1. Partitioning
  2. Communication
  3. Agglomeration
  4. Mapping

$\left.\begin{array}{l}\text{Partitioning}\\ \text{Communication}\end{array}\right\}$ problem specific

$\left.\begin{array}{l}\text{Agglomeration}\\ \text{Mapping}\end{array}\right\}$ system specific

(available online: http://www.mcs.anl.gov/~itf/dbpp)

Partitioning

- Fine-grained decomposition into primitive tasks
  - Task (or functional) decomposition
    - Decompose based on central computation
    - What data is needed to perform a task?
  - Data (or domain) decomposition
    - Decompose based on central data structures
    - How can operations on decomposed data structures be realized?
- Always consider both and different variants of them
- Decomposition gives upper bound on parallelism
- Goals
  - A lot more primitive tasks than processing/memory elements
  - Number of tasks scales with the problem size

Communication

- Dependency analysis between primitive tasks
  - Data or temporal dependencies

- Typical properties
  - Local or global dependencies
    - i. e., communicate with few or many
  - Structured or unstructured
    - Structures can be taken advantage of later
  - Static or dynamic communication partners
  - Synchronous or asynchronous
    - Do we now beforehand what data is needed where?

- Goals for communication patterns
  - Local, balanced, concurrent

Agglomeration

- Group primitive tasks into larger tasks
  - Reduce dependencies and overhead, increase locality and concurrency
  - Trade off computation and communication
    - Replication of data and/or computation?
- Goals
  - Structure algorithm towards real system
  - Retain scalability
  - Balance computation and communication (if beneficial)
    - Especially in one-task-per-processor situations

Mapping

- Map agglomerated tasks to processing/memory elements
  - Externally: Create one process/thread per task
    - Mapping handled by the operating system
  - Internally: Handle multiple tasks per thread/process

- Goals
  - Minimized execution time
    - Overlap communication with computation
  - Load balancing

Common Algorithm Structures

- Organized by tasks
  - Task parallelism
  - Divide and Conquer
- Organized by data decomposition
  - Geometric decomposition
  - Recursive data
- Organized by data flow
  - Pipeline
  - Event-based coordination

Task parallelism

- Many tasks allow balanced scheduling
  - Useful when tasks complexity varies or the system is heterogeneous in some way
  - Rule of thumb: have at least 10 times more tasks than processing elements (but still consider management overhead)
  - Task scheduling
    - Static vs. dynamic
    - Centralized vs. decentralized work queue
- Embarrassingly parallel tasks
  - No dependencies between tasks

Divide and Conquer

- Principle
  - Split problem into subproblems recursively
  - Solve primitive subproblems
  - Combine partial solutions to one solution recursively
- Recursive generation of tasks
  - Fall back to sequential solution at some point
- Does not fully utilize system at the beginning and the end
  - Divide into more than two subproblems
  - Combine more than two partial solutions
  - Divide/combine in parallel

Geometric Decomposition

- Split a regular data structure into regular chunks
  - Blocked and cyclic layouts along one or more dimensions

1D block

P0
P1
P2
P3

2D block

P0	P1
P2	P3

2D block-cyclic

P0	P1	P0	P1
P2	P3	P2	P3
P0	P1	P0	P1
P2	P3	P2	P3

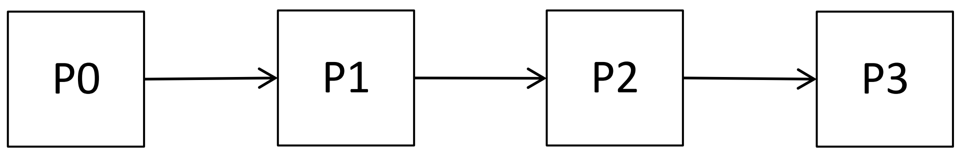
- Cyclic layouts enable static load balancing
- Dynamic assignment of chunks also possible

Recursive Data

- Seemingly sequential operations on recursive data structures
  - e. g., linked lists, trees, graphs, …
- Divide & Conquer works in some cases
- Rethink the problem in terms of operations on every member (e. g. recursive doubling)
  - Might increase algorithmic complexity, but could still be faster for a certain number of processors
  - Might work well on SIMD architectures

Pipeline

- For simple ordering constraints
- Throughput oriented



- Concurrency limited by the number of stages
- Slowest stage might become a bottleneck
  - Parallelize stages
  - Have multiple pipelines

Event-Based Coordination

- For irregular, dynamic ordering constraints
  - “Generalized pipeline”
- Tasks wait for events, process them and generate events themselves
- Requires one or multiple event queues

More complex problems

- Break it down into manageable pieces
  - Express problem in terms of known problems
  - Apply existing algorithms
  - Design missing pieces (e. g., with Foster's methodology)

- Benefit from work of others
  - Libraries with already parallelized primitives
    - Optimized and improved over (a long) time
  - Remaining task: reorganize data between function calls