Distributed and Parallel Technology

Revision

Hans-Wolfgang Loidl

School of Mathematical and Computer Sciences
Heriot-Watt University, Edinburgh

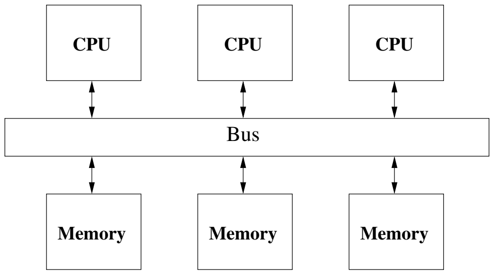
HERIOT WATT UNIVERSITY

Semester 2 — 2016/17

Table of Contents

Classes of Architectures

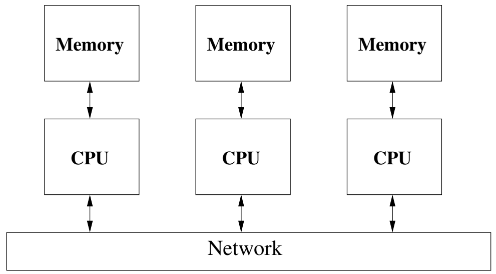
- *Shared memory:* CPUs access common memory across high-speed bus



- Symmetric Multi-Processing (SMP), e.g. Sun SMP
  - advantage: very fast communication between processors
  - disadvantage: bus contention limits number of CPUs
- hierarchical SMP, e.g. IBM ASCI White, 1.512 * 16 PowerPC SMP

Classes of Architectures

- *Distributed memory:* CPUs communicate by message passing on dedicated high-speed network (e.g. IBM SP2, Cray T3E)



-
  - advantage: highly scalable
  - disadvantage: explicit data communication is relatively slow

Parallel Architectures & Clusters

- The major distinction is between:
  - Single Instruction Multiple Data (*SIMD*);
  - Multiple Instruction Multiple Data (*MIMD*)
- SIMD typically involves specialised CPU & communications
  - Control CPU + multiple ALUs e.g. CDC 6600
  - Today's graphics processors (GPGPUs)
- MIMD typically involves specialised communications
  - Point to point on channels e.g. Meiko Computing Surface
  - Communication hierarchy e.g. nCube, BBN Butterfly

Parallel Architectures & Clusters

Parallel hardware is increasingly *heterogeneous*:

- Often SIMD components complement von Neumann CPUs in standard microprocessors
  - digital signal processing (DSP) on vectors of bits
  - mainly for graphics and animation
    e.g. NVidias Tesla cards or Intel MMX instructions
- poor support in compilers: the programmer must drop into assembly language
- no generic libraries: compiler specific

Parallel Programming Languages

More recent parallel programming languages offer increasing *levels of abstraction* to simplify parallel programming:

- *Very Low level:* sequential host language (eg. C) with basic primitives for coordination (semaphores, sockets);
  - Advantage: complete control over coordination; potentially very high performance
  - Disadvantage: very difficult to program; error-prone
- *Low level:* sequential host language (eg. C, Fortran) with a library for communication and coordination (*C+MPI*);
  - Advantage: standardised; complete control over coordination; potentially very high performance;
  - Disadvantage: difficult to program; error-prone
  - Notes: very well suited for *clusters*

Parallel Programming Languages (cont'd)

- *Mid level:* sequential host language (eg. C, Fortran) with compiler directives for parallelism (*OpenMP*);
  - Advantage: easy to introduce parallelism, if the structure fits (eg. data-parallelism);
  - Disadvantage: less flexible than general approaches; not all programs fit the structure
  - Notes: very well suited for *multi-cores*

Parallel Programming Languages (cont'd)

- *High level:* extension of a sequential host language (eg. C) with constructs for coordinating parallelism and distributing data (*UPC* and other *PGAS* languages);
  - Advantage: conceptually simple, because PGAS gives the illusion of a (globally) shared memory
  - Disadvantage: data placement and performance tuning is tricky; poor pointer safety
- *Very high level:* alternative languages with (semi-)implicit parallelism (*GpH*, *SAC*);
  - Advantage: introducing parallelism is simple and doesn't change the result (deterministic);
  - Disadvantage: performance tuning is tricky; implementation overhead; language is unfamiliar
  - Notes: abstracts over harwdare structure; very well suited for *symbolic computation*

Basic Point to Point Communication in MPI

MPI offers two basic point to point communication functions:

- `MPI_Send(message, count, datatype, dest, tag, comm)`
  - Blocks until `count` items of type `datatype` are sent from the `message` buffer to processor `dest` in communicator `comm`.
    - ★ `message` buffer may be reused on return, but message may still be in transit!
- `MPI_Recv(message, count, datatype, source, tag, comm, status)`
  - Blocks until receiving a `tag`-labelled message from processor `source` in communicator `comm`.
  - Places the message in `message` buffer.
    - ★ `datatype` must match datatype used by sender!
    - ★ Receiving fewer than `count` items is OK, but receiving more is an error!

Aside: These are the two most important MPI primitives you have to know.

Send and Receive in more Detail

```
int MPI_Send(
  void * message,
  int count,
  MPI_Datatype datatype,
  int dest,
  int tag,
  MPI_Comm comm)
```

```
int MPI_Recv(
  void * message,
  int count,
  MPI_Datatype datatype,
  int source,
  int tag,
  MPI_Comm comm,
  MPI_Status * status)
```

- `message` pointer to send/receive buffer
- `count` number of data items to be sent/received
- `datatype` type of data items
- `comm` communicator of destination/source processor
  - For now, use default communicator `MPI_COMM_WORLD`
- `dest/source` rank (in `comm`) of destination/source processor
  - Pass `MPI_ANY_SOURCE` to `MPI_Recv()` if source is irrelevant
- `tag` user defined message label
  - Pass `MPI_ANY_TAG` to `MPI_Recv()` if tag is irrelevant
- `status` pointer to struct with info about transmission
  - Info about source, tag and #items in message received

High Level Parallel Programming

Many approaches have been proposed to reduce the programmer's coordination management burden, e.g. skeletons, parallelising compilers, etc.

GpH (Glasgow parallel Haskell) aims to simplify parallel programming by requiring the programmer to specify only a few key aspects of parallel programming, and leaving the language implementation to automatically manage the rest.

GpH is a parallel extension to the non-strict, purely functional language Haskell.

What are the basic primitives to introduce parallelism; what is their semantics?

GpH Coordination Aspects

To specify parallel coordination in Haskell we must

1. Introduce parallelism
2. Specify Evaluation Order
3. Specify Evaluation Degree

This is much less than most parallel paradigms, e.g. no communication, synchronisation etc.

It's important that we do so without cluttering the program. In many parallel languages, e.g. C with MPI, coordination so dominates the program text that it obscures the computation.

Evaluation Strategies: Separating Computation and Coordination

Evaluation Strategies abstract over `par` and `pseq`,

- raising the level of abstraction, and
- separating coordination and computation concerns
  It should be possible to understand the semantics of a function without considering its coordination behaviour.

How can you implement a data-parallel strategy over a list?

Parallel Performance Tuning

Consider:

- How to use thresholding in divide-and-conquer programs?
- How to use chunking in data-parallel programs?
- How to code these techniques in parallel Haskell?

Go through the *worked example* of parallelisation and tuning from the GpH slides!

Evaluation Strategy Summary

Critically evaluate the properties of strategies.
Evaluation Strategy

- use laziness to separate algorithm from coordination
- use the `Eval` monad to specify evaluation order
- use overloaded functions (`NFData`) to specify the evaluation-degree
- provide high level abstractions, e.g. parList, parSqMatrix
- are functions in algorithmic language ⇒
  - comprehensible,
  - can be combined, passed as parameters etc,
  - extensible: write application-specific strategies, and
  - can be defined over (almost) any type
- general: pipeline, d&c, data parallel etc.
- Capable of expressing complex coordination, e.g. Embedded parallelism, `Clusteri`ng, skeletons

A Methodology for Parallelisation

1. *Sequential implementation.* Start with a correct implementation of an inherently-parallel algorithm.
2. *Parallelise Top-level Pipeline.* Most non-trivial programs have a number of stages, e.g. lex, parse and typecheck in a compiler. Pipelining the output of each stage into the next is very easy to specify, and often gains some parallelism for minimal change.
3. *Time Profile* the sequential application to discover the “big eaters”, i.e. the computationally intensive pipeline stages.
4. *Parallelise Big Eaters* using evaluation strategies. It is sometimes possible to introduce adequate parallelism without changing the algorithm; otherwise the algorithm may need to be revised to introduce an appropriate form of parallelism, e.g. d & c or data-parallelism.

A Methodology for Parallelisation

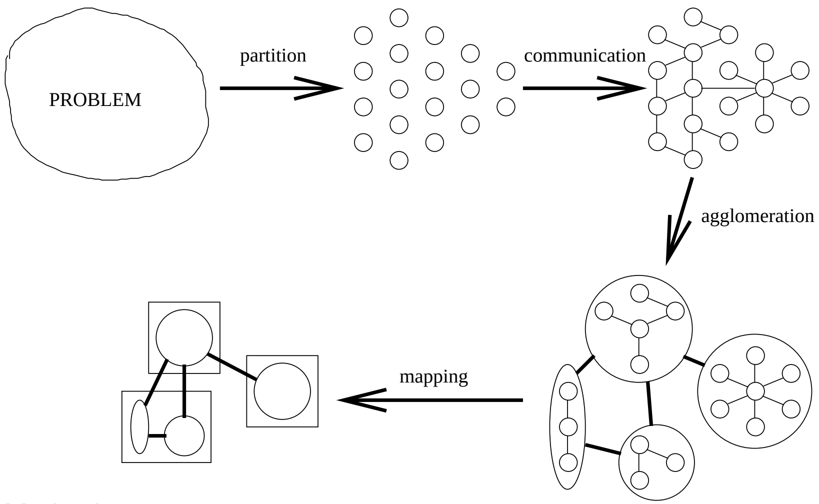
1. *Idealised Simulation.* Simulate the parallel execution of the program on an idealised execution model, i.e. with an infinite number of processors, no communication latency, no thread-creation costs etc. This is a “proving” step: if the program isn’t parallel on an idealised machine it won’t be on any real machine. A simulator is often easier to use, more heavily instrumented, and can be run in a more convenient environment, e.g. a desktop.
2. *Realistic Simulation.* Some simulators, like GranSim, can be parameterised to emulate a particular parallel architecture, forming a bridge between the idealised and real machines. A major concern at this stage is to improve thread granularity so as to offset communication and thread-creation costs.
3. *Tune on Target Architecture.* Use performance visualisation tools (generally less detailed) to improve performance.

At the latter 3 stages, consider alternative parallelisations.

General issues of performance tuning

- Consider the *granularity* of the parallelism: how much work is done in each thread?
- Consider the *computation-communication ratio*
- Avoid communication as much as possible, or try to overlap communication with computation (*latency hiding*)
- Measure run-time and speed-up to assess the quality of the code
- Distinguish between relative speed-up (parallel vs parallel 1-processor code) and absolute speed-up (parallel vs sequential code)
- Assess performance through speedup and scalability graphs
- Assess issues beyond performance: programmability, portability, ease-of-maintenance

Foster’s PCAM Parallel Program Design Methodology



Methodology

- see DBPP Online, Part I, Chapter 2

Algorithmic Skeletons — The Computation Model

A *skeleton* is

- a useful pattern of parallel computation and interaction,
- packaged as a *framework/second order/template* construct (i.e. parametrised by other pieces of code).
- *Slogan:* Skeletons have *structure* (coordination) but lack *detail* (computation).

Each skeleton has

- one interface (e.g. generic type), and
- one or more (architecture-specific) implementations.
  - Each implementations comes with its own *cost model*.

A skeleton *instance* is

- the code for computation together with
- an implementation of the skeleton.
  - The implementation may be shared across several instances.

Note: Skeletons are more than design patterns.

Skeletons Are Parallel Higher-Order Functions

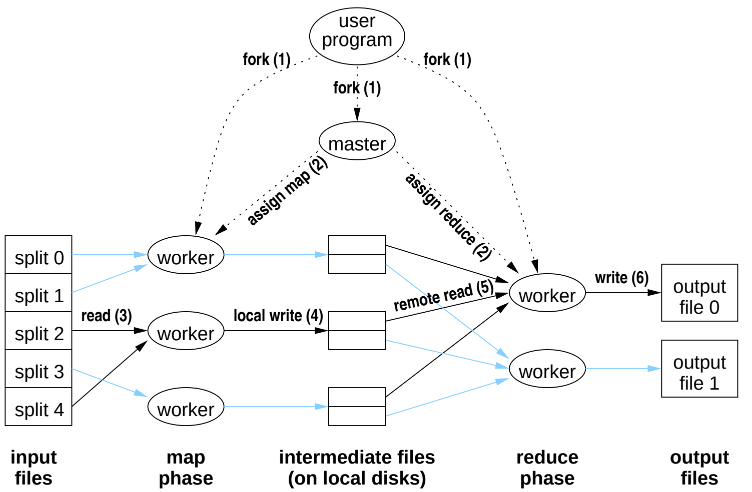
Observations:

- A *skeleton* (or any other template) is essentially a higher-order function (HOF), ie. a function taking functions as arguments.
  - Sequential code parameters are functional arguments.
- Skeleton implementation is parallelisation of HOF.
- Many well-known HOFs have parallel implementations.
  - Thinking in terms of higher-order functions (rather than explicit recursion) helps in discovering parallelism.

Consequences:

- Skeletons can be combined (by function composition).
- Skeletons can be nested (by passing skeletons as arguments).

A Skeleton Implementation — Google MapReduce



- J. Dean, S. Ghemawat. *MapReduce: Simplified Data Processing on Large Clusters*, Commun. ACM 51(1):107–113, 2008