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

- **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. NVIDIA 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 (e.g. 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 (e.g. 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 (e.g. C, Fortran) with compiler directives for parallelism (OpenMP);
  - Advantage: easy to introduce parallelism, if the structure fits (e.g. 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 (e.g. 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 hardware structure; very well suited for symbolic computation

Send and Receive in more Detail

```c
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
- **dest/source** rank (in comm) of destination/source processor
- **tag** user defined message label
- **status** Info about source, tag and #items in message received

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.

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

- Introduce parallelism
- Specify Evaluation Order
- 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 \( \text{par} \) and \( \text{pseq} \),

- raising the level of abstraction, and
- separating coordination and computation concerns

\textit{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 \textit{worked example} of parallelisation and tuning from the GpH slides!

Evaluation Strategy Summary

\textit{Critically evaluate} the properties of strategies.

Evaluation Strategy

- use laziness to separate algorithm from coordination
- use the \textit{Eval} monad to specify evaluation order
- use overloaded functions (\textit{NFData}) to specify the evaluation-degree
- provide high level abstractions, e.g. \textit{parList}, \textit{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, \textit{d&c}, data parallel etc.
- Capable of expressing complex coordination, e.g. Embedded parallelism, \textit{Clustering}, skeletons
Foster’s PCAM Parallel Program Design Methodology

Algorithmic Skeletons — The Computation Model

Methodology

- see DBPP Online, Part I, Chapter 2

Skeletons Are Parallel Higher-Order Functions

Observations:
- A skeleton (or any other template) is essentially a higher-order function (HOF), i.e., 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

Hadoop HDFS

Hadoop HDFS is a high-performance, distributed file system, used with Hadoop to efficiently manage large volumes of data in parallel applications.

Consider:

- What are the design goals of HDFS?
- What are the main design choices in implementing HDFS?
- What is the role of HDFS inside the Hadoop infrastructure?
- What is the network topology for HDFS?
- What is the logical architecture of HDFS?
- Examples on how HDFS deals with faults.