Part I: Parallel Program Design

We follow
  - DBPP Online: http://www.mcs.anl.gov/~itf/dbpp/

Parallel Programming Model — Tasks

- Parallel computation = set of concurrently executing tasks
- Task = sequential program + local memory + inports + outports
- Tasks can
  - compute (in local memory),
  - send messages to outports or receive messages from inports,
  - create new tasks or terminate.
  - Tasks can vary dynamically during program execution.
- Multiple tasks can be mapped to physical processors.
  - Mapping does not affect the program semantics.

Model: *tasks* connected by *channels*
- well-suited for distributed memory architectures (and MPI)
- see DBPP Online, Part I, Chapter 1.3
Parallel Programming Model — **Channels**

- Channel = outport (task $T_1$) + message queue + inport (task $T_2$)
  - Channels are *uni-directional* (from $T_1$ to $T_2$)
  - Messages are *ordered* (FIFO order)
- Communication topology can vary dynamically.
  - Channels can be created and deleted.
  - References to channels (ports) can be sent in messages, i.e. channels are *first class objects*.

Designing Parallel Algorithms — **Partition**

**Goal:** Identify parallel tasks — *the more the better.*

**Methods:**
- Domain decomposition: divide data
  - E.g. matrix decomposition
- Functional decomposition: divide computation
  - E.g. pipeline

**Good Design** checklist:
- Does #tasks scale with problem size?
  - Or else algorithm will not scale.
- Tasks of comparable size?
  - Or else load balancing will be hard.
- Does partition avoid redundant computation/storage?
  - Or else algorithm may not scale.
- #tasks > 10 * #processors?
  - Or else there will not be much left to design in later stages.
- Are there alternative partitions?

Designing Parallel Algorithms — **Communication**

**Goal:** Identify channels — *the less the better.*

**Guidelines:**
- Prefer *local* over *global* communication.
  - Distribute global comm via divide-and-conquer (e.g. merge sort)
- Compute *and communicate* concurrently (*latency hiding*).
  - Re-order computation and communication.
  - Consider asynchronous (request/response) communication.

**Good Design** checklist:
- All tasks perform about the same number of comm operations?
  - If not try to distribute comm operations more equitably.
- Each task communicates only with few neighbours?
  - If not try to distribute global communication.
- Are communication operations able to proceed concurrently?
  - If not try to parallelise using divide-and-conquer.
- Are tasks able to compute concurrently?
  - If not try reordering communication and computation, or try a different algorithm.
Designing Parallel Algorithms — **Agglomeration**

**Goal:** Combine tasks — to *improve performance* or reduce *devel cost*. 

**Guidelines:**
- **Maintain scalability while**
  - Increasing *locality*
    - By grouping senders and receivers of data together.
    - By replicating data/computation.
  - Decreasing *granularity*
    - By changing domain decomposition.
  - Re-using *sequential code*

**Note:** Agglomeration will in general yield more tasks than processors.
- If \(#\text{tasks} = \#\text{processors}\) skip *Mapping* step.

---

Designing Parallel Algorithms — **Mapping**

**Goal:** Assign tasks to procs — *maximise utilisation & minimise comm*

**Methods:**
- **Static task allocation:**
  - 1 task/proc is optimal for regular computation cost
  - E.g. parallelisation by domain decomposition
- **Dynamic task allocation (aka. scheduling):**
  - For programs with irregular computation cost, irregular communication irregular, or dynamically variable \(#\text{tasks}\).
  - Requires order of magnitude more tasks than processors.
- **Centralised allocation:**
  - Master sends tasks to fixed pool of workers.
  - **Note:** Good locality but master can become bottleneck.
- **Distributed allocation:**
  - Each processor may *pull* tasks from or *push* tasks to neighbours.
  - Strategy (who to push/pull from) may be probabilistic.
  - **Note:** Good scalability but hard to maintain locality.

**Good Design** checklist:
- Is communication cost reduced and locality increased?
- Does benefit of replicated computation outweigh cost?
- Does replicated data compromise scalability?
  - E.g. not scalable if replicated data grows linearly in \(#\text{processors}\).
- Are tasks similar in computation and comm costs?
- Does \(#\text{tasks}\) still scale with problem size?
- Is there still sufficient parallelism?
  - Beware: Ultimate goal is efficiency, not maximum parallelism!
- Could tasks be agglomerated further?
  - Other things being equal, coarser granularity increases efficiency.
- Development cost of modifying existing sequential code?
Part II: Parallel Programming
Design Patterns

We follow
T. Mattson, B. Sanders, B. Massingill. Patterns for Parallel Programming, Addison-Wesley, 2005
- Complement's Foster's approach
- Implements parallel patterns in C#

Finding Concurrency

3 classes of patterns:
- Decomposition Patterns
  - Task decomposition: decomp problem into concurrent tasks
  - Data decomposition: decomp data into independent units
- Dependency Analysis Patterns
  - identify task dependencies (emphasis on data sharing)
  - group/order tasks
- Design Evaluation
  - not a pattern
  - similar to Foster's good design checklists

Algorithm Structure

3 classes of patterns:
- Organise by Task Decomposition
  - Linear: Task Parallelism
  - Recursive: Divide & Conquer
- Organise by Data Decomposition
  - Linear: Geometric Decomposition
    - E.g. lists, vectors, matrices
  - Recursive: Recursive Data
    - E.g. trees
- Organise by Data Flow
  - Regular: Pipeline or task DAG
    - static communication structure
  - Irregular: Event-based co-ordination
    - dynamic (often unpredictable) communication structure
Part III: Algorithmic Skeletons

Resources:
  - http://homepages.inf.ed.ac.uk/mic/Pubs/skeletonbook.ps.gz
- Skeletal Parallelism homepage

Algorithmic Skeletons — What?

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.

Algorithmic Skeletons — How and Why?

Programming methodology:
- Write sequential code, identifying where to introduce parallelism through skeletons.
- Estimate/measure sequential processing cost of potentially parallel components.
- Estimate/measure communication costs.
- Evaluate cost model (using estimates/measurements).
- Replace sequential code at sites of useful parallelism with appropriate skeleton instances.

Pros/Cons of skeletal parallelism:
- simpler to program than unstructured parallelism
- code re-use (of skeleton implementations)
- structure may enable optimisations
  - *not* universal

Common Skeletons — Pipeline

- Data flow skeleton
  - Data items pass from stage to stage.
  - All stages compute in parallel.
  - Ideally, pipeline processes many data items (e.g. sits inside loop).
Pipeline — Load Balancing

Typical problems:
- Ratio communication/computation too high.
- Computation cost not uniform over stages.

Ad (1) Pass chunks instead of single items

Ad (1,2) Merge adjacent stages

Common Skeletons — Parallel Tasks

Data flow skeleton
- Input split on to fixed set of (different) tasks.
- Tasks compute in parallel.
- Output gathered and merged together.
  - Split and merge often trivial; often executed on proc 1.

- Dual (in a sense) to pipeline skeleton.

Beware: Skeleton name non-standard.

Common Skeletons — Task Farm

Data parallel skeleton (e.g. parallel sort scatter phase)
- Farmer distributes input to a pool of $N$ identical workers.
- Workers compute in parallel.
- Farmer gathers and merges output.

Static vs. dynamic task farm:
- Static: Farmer splits input once into $N$ chunks.
- Dynamic: Farmer continually assigns input to free workers.

Task Farm — Load Balancing

Typical problems:
- Irregular computation cost (worker).
  - Use dynamic rather than static task farm.
  - Decrease chunk size: Balance granularity vs. comm overhead.

- Farmer is bottleneck.
  - Use self-balancing chain gang dynamic task farm.
    - Workers organised in linear chain.
    - Farmer keeps track of # free workers, sends input to first in chain.
    - If worker busy, sends data to next in chain.
Common Skeletons — Divide & Conquer

Recursive algorithm skeleton (e.g. parallel sort merge phase)

Skeletons in the Real World

Skeletal Programming

- can be done in many programming languages,
  - skeleton libraries for C/C++
  - skeletons for functional languages (GpH, OCaml,...)
  - skeletons for embedded systems

- is still not mainstream,

- but an active area of research.
  - > 30 groups/projects listed on skeleton homepage

- and it is slowly becoming mainstream
  - TPL library of Parallel Patterns in C# (blessed by Microsoft)

Common Skeletons — Divide & Conquer II

Recursive algorithm skeleton

- Recursive call tree structure
  - Parent nodes *divide* input and pass parts to children.
  - All leaves compute the same sequential algorithm.
  - Parents gather output from children and *conquer*, i.e. combine and post-process output.

To achieve good load balance:

- Balance call tree.
- Process data in parent nodes as well as at leaves.

Part IV: Implementing Skeletons
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).

Skeletons Are PHOFs — Pipeline

\[
\begin{align*}
  x & \rightarrow g \\
  g & \rightarrow f(g(x))
\end{align*}
\]

Code (parallel implementation in red)
\[
\text{pipe2} :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c \\
\text{pipe2} f g x = \text{let } y = g x \text{ in } y \mathbin{\text{'par'}} f y
\]

Notes:
- \text{pipe2} is also known as function composition.
- In Haskell, sequential function composition is written as \text{.} (read “dot”).

Skeletons Are PHOFs — Parallel Tasks

\[
\begin{align*}
x & \rightarrow \text{split} \\
\text{split} & \rightarrow f \rightarrow g \\
f & \rightarrow f y \\
g & \rightarrow g z \\
\text{merge} & \rightarrow \text{merge}(f y, g z)
\end{align*}
\]

Code (parallel implementation in red)
\[
\text{task2} :: (a \rightarrow (b,c)) \rightarrow (c \rightarrow e) \rightarrow (a \rightarrow f) \\
\text{task2} \text{split merge f g x = let } (y,z) = \text{split x} \\
f y & = f y \\
g z & = g z \text{ in} \\
fy \mathbin{\text{'par'}} gz \mathbin{\text{'pseq'}} \text{merge fy gz}
\]

Skeletons Are PHOFs — Task Farm

\[
\begin{align*}
[x_1, x_2, ..., x_N] & \rightarrow \text{map } f \\
\text{map } f & \rightarrow [f x_1, f x_2, ..., f x_N] \\
x_1 & \rightarrow f x_1 \\
x_2 & \rightarrow f x_2 \\
... & \\
x_N & \rightarrow f x_N
\end{align*}
\]

Code (parallel implementation in red)
\[
\text{farm} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b] \\
\text{farm } f \text{ [] } = [] \\
\text{farm } f \text{ (x:xs)} = \text{let } fx = f x \text{ in} \\
fx \mathbin{\text{'par'}} fx : (\text{farm } f \text{ xs})
\]

Notes:
- \text{farm} is also known as parallel map.
  - Map functions exist for many data types (not just lists).
- Missing in implementation: strategy to force eval of lazy list.
- Strategies also useful to increase granularity (by chunking).
Skeletons Are PHOFs — Divide & Conquer

**Notes:**
- Divide & Conquer is a generalised parallel fold.
  - Folds exist for many data types (not just lists).
  - Missing in impl: strategies to force eval and improve granularity.
- **Aside:** folding/reducing lists
  ```
  fold :: (a -> a -> a) -> a -> [a] -> a
  -- fold f e [x1,x2,...,xn] == e 'f' x1 'f' x2 ... 'f' xn, provided that
  -- (1) f is associative, and
  -- (2) e is an identity for f.
  -- Tail-recursive sequential implementation:
  fold f e [] = e
  fold f e (x:xs) = fold f (e 'f' x) xs
  -- Parallel implementation as instance of divide & conquer:
  fold f e = dnc split f atomic evalAtom where
  split xs = splitAt (length xs 'div' 2) xs
  atomic [] = True
  atomic [_] = True
  atomic _ = False
  evalAtom [] = e
  evalAtom [x] = x
  ```

Program Transformations

**Observation:**
- HOFs can be transformed into other HOFs with provably equivalent (sequential) semantics.
- **Example:** Pipeline of farms vs. farm of pipelines
  ```
  map g . map f == map (g . f)
  ```
  ```
  farmer
  f
  g
  farmer
  ```
- **use** map g . map f (pipe of farms) if ratio comp/comm high
- **use** map (g . f) (farm of pipes) if ratio comp/comm low
- **More transformations in**

Program Development with Functional Skeletons

**Programming Methodology:**
- Write seq code using HOFs with known equivalent skeleton.
- Measure sequential processing cost of functions passed to HOFs.
- Evaluate skeleton cost model.
- If no useful parallelism, transform program and go back to 3.
- Replace HOFs that display useful parallelism with their skeletons.

**Tool support:**
- Compilers can automate some steps (see Michaelson/Scaife)
  - Only for small, pre-selected set of skeletons
- **Example:** PMLS (developed by Greg Michaelson et al.)
  - **Skeletons:** map/fold (arbitrarily nested)
  - **Automates steps 2-5.**
    - Step 2: automatic profiling
    - Step 4: rule-driven program transformation + synthesis of HOFs
    - Step 5: map/fold skeletons implemented in C+MPI
Further Reading

  Online: http://www.mcs.anl.gov/~itf/dbpp/

  Online: http://dx.doi.org/10.1145/1327452.1327492

- G. Michaelson, N. Scaife. “Skeleton Realisations from Functional Prototypes”, Chap. 5 in S. Gorlatch and F. Rabhi (Eds), Patterns and Skeletons for Parallel and Distributed Computing, Springer, 2002

  ISBN10: 0124159931 (paperback)