What are Design Patterns?

- Design patterns are templates for “solving” a problem
  - Solving = (structuring, executing, writing, analysing, …)
  - Range from application level down to scheduling instructions
  - Emphasis on **composability**: able to safely compose components
- Structured programming introduced the first design patterns
  - Ordering execution of statements: sequences, branches, loops
  - Grouping data into a unit: structures
  - Grouping statements into a unit: procedures and functions
- Object-oriented programming standardised common patterns
  - `struct X; F(X*,int);` → `class X{  F(int);  }`
  - Polymorphism (virtual functions) increase composability
  - Don’t need to know what an object is, just what it does

Describing Design Patterns

- Design patterns try to formally capture intuition or experience
  - Reduce the need for super-star programmers: engineering not art
  - Increase productivity: don’t re-invent the wheel
  - Increase reliability: record both the patterns and when they apply
- Elements of a design pattern
  - **Name**: we need a common name so people can talk about them
  - **Problem**: simple description of what problem the pattern solves
  - **Context**: where does the problem occur, and any background info
  - **Forces**: any intrinsic tradeoffs that are being addressed
  - **Solution**: how the pattern is applied
  - **Examples**: application of the pattern to a real-world example

Application of Design Patterns

- Concreteness of the design pattern varies hugely
  1. Both solution and conditions are described programmatically
     - Can be turned into compiler optimisations
     - Rare and usually local in scope: checking conditions is difficult
     - *Why does this course even exist* – *can’t the compiler do it?*
  2. Solution can be captured in code, but not the conditions
     - Compiler/library will apply the pattern for us: `spawn/parallel_for`
     - We need to check whether the pattern can and should be applied
  3. Both solution and conditions cannot be formally captured
     - Programmer has to interpret the conditions and the solution
     - Requires human skill and (maybe) some thought

Umm...

- This all sounds great in theory, but the practise is less good
  - People have different names for the same thing
  - People have the same name for different things
  - Currently a lack of standardisation in parallel pattern names
  - GPUs etc. are muddying the waters further
- I will broadly use the terms from these two sources
  - Berkely parlab: [http://parlab.eecs.berkeley.edu/wiki/patterns/patterns](http://parlab.eecs.berkeley.edu/wiki/patterns/patterns)
Map-Reduce

- **Problem**: a function must be applied to many pieces of data, followed by an associative reduction

- **Context**: some container contains $x_1..x_n$, and we wish to calculate $a( f(x_1), a(f(x_2), ... f(x_n)) )$. Both $f(.)$ and $a(.)$ must be side-effect free, and $a(.)$ must be associative

- **Solution**: apply $f(.)$ to all data-items as independent parallel tasks (map), then use a recursive tree of parallel tasks to collect the results (reduce)

```
double max_magnitude(int n, const complex_t *data) {
    double best = abs(data[0]); // potential bug?
    for(int i=1;i<n;i++) {
        double curr = abs(data[i]);
        if(curr > best)
            best = curr;
    }
    return best;
}
```

* f(.) = abs(.) - Complex magnitude (modulus)
* a(.) = max(.) – Maximum of two numbers
* Do we meet requirements of pattern?
  * $a(.)$ and $f(.)$ are side-effect free
  * $a(.)$ is associative

```
double max_magnitude(int n, const complex_t *data) {
    std::vector<complex_t> temp(n);
    parallel_for(i=0:n-1){ // pseudo-syntax
        temp[i] = abs(data[i]);
    }
    for(int i=0;i<n;i++) {
        if(temp[i] > best)
            best = temp[i];
    }
    return best;
}
```
double max_magnitude(int n, const complex_t *data) {
    if(n==0){
        return abs(data[0]);
    }else{
        double left=spawn max_magnitude(n/2, data);
        double right=spawn max_magnitude(n-n/2, data+n/2);
        sync;
        return max(left,right);
    }
}

Map-Reduce: Forces

- Balancing parallelism vs overhead
  - We want to maximise available tasks to increase parallelism
  - But the reduction graph contains lots of synchronisation

- Determinism vs speed
  - Want to have as much scheduling flexibility as possible
  - Prefer deterministic results with pseudo-associative operations
    - e.g. floating-point addition

(“I find the “Forces” section often ends up a bit vague”)

Implementation of Map-Reduce

- Map-Reduce occurs everywhere
  - Sequential version is available in many languages and libraries
  - Often the map and reduce functions must be combined
    - \( m(x,y) = a(x, m(y)) \)
  - C++: std::accumulate; Python: reduce; Haskell: foldl/foldr
- Parallel versions of Map-Reduce are very common
  - Very easy to understand for users
  - Applicable in large numbers of real-world scenarios
- Google use a tool called MapReduce with thousands of CPUs
  - Excellent paper on real-world scaling in distributed systems:
    - [http://research.google.com/archive/mapreduce.html](http://research.google.com/archive/mapreduce.html)
  - But they didn’t invent the idea, been around for decades
  - Many similar approaches: Hadoop etc.
struct Worker {
  double m_max;
  const complex_t *m_data;
  Worker(const complex_t *data) {
    m_data = data;
    m_max = 0.0;
  }
  Worker(Worker &src, tbb::split) {
    m_data = src.m_data;
    m_max = 0.0;
  }
  template<class T>
  void operator()(const T &r) {
    double acc = 0.0;
    for (int i = r.begin(); i < r.end(); i++)
      acc = std::max(acc, abs(m_data[i]));
    m_max = std::max(m_max, acc);
  }
  void join(Worker &rhs) {
    m_max = std::max(m_max, rhs.m_max);
  }
};

Element-wise / Data-Parallel

- **Problem**: apply a function to all items in a container
- **Context**: an identical function is to be applied to all items in a container, either transforming the items themselves, or performing some other side-effect
- **Forces**: standard (no. of tasks vs task overhead)
- **Solution**: create work by sub-dividing the container
- **Examples**: tbb::parallel_for, CUDA, OpenCL

Agglomeration

- **Problem**: an application has too much fine-grain parallelism
- **Context**: many algorithms can be easily decomposed to the level where each task executes a single instruction, but this results in an extremely high cost of work
- **Forces**: we need to balance the need for average parallelism versus the cost of work
- **Solution**: halt production of new tasks when the remaining work drops below a certain level, and switch to sequential execution
### Agglomeration Examples

- **Recursive Matrix-Matrix Multiply**
  - Switching to sequential when the matrix gets too small
  - Potential problem: a fixed threshold may give sub-optimal results
    - What is optimal on one platform may not be on another
    - Compiler optimisations may increase the agglomeration required

- `tbb::parallel_for` and `tbb::parallel_reduce`
  - Function object is given a range of indices, not a single index
  - Size of range is optimised to try to balance execution time
  - Potential problem: auto-optimisation may guess wrong
    - Splits a large but extremely fast range into individual tasks
    - Doesn't split a small loop where each iteration is very slow
  - Usually auto-tuning is better, unless you perform experiments
    - Humans are pretty bad at guessing how fast things run
    - 8-Cpus used to be rare, now 32-Cpus is fairly common

### Divide and Conquer / Recursive

- **Problem**: a sequential program with recursive functions or data-structures must be parallelised
- **Context**: the existing program repeatedly splits the data into independent sub-sections, which can then be further decomposed. The splitting may be over a 1D or 2D range, or follow some graph-like data-structure
- **Forces**: standard (creating parallelism vs task size)
- **Solution**: when the program splits the problem, spawn new tasks for each sub-problem, then sync for all child tasks

```c
// Split the range in two, returning // the pivot element. Takes // O(end-begin) steps
double *partition(
    double *begin,
    double *end
);

void QSort(
    double *begin,
    double *end
){
    if(end-begin<1000){
        // Agglomeration - drop to serial
        std::sort(begin, end);
    }else{
        double *mid=partition(begin, end);
        spawn QSort(begin, mid-1);
        spawn QSort(mid, end);
    }
}
```

- **Sorting**: fundamental operation
  - Quick-Sort repeatedly splits problem into smaller problems
  - What is the big-O complexity?
- Sorting is so common it is supported directly in TBB
  - `tbb::parallel_sort`
  - Same interface as `std::sort`
  - Guaranteed deterministic
    - Always gives the same result
    - Not guaranteed `stable`
      - Different objects with the same key may be re-ordered

### Pipeline

- **Problem**: a stream of data needs to be processed in stages
- **Context**: some source function produces a stream of individual chunks of data, which must then be transformed using a set of operations, then sent to some sink function
- **Forces**: allowing many parallel data items increases available parallelism, but can greatly increase memory requirements
- **Solution**: create a pipeline of filter objects which can be applied as parallel tasks, then manage lifecycle of a limited number of tokens through the pipeline
Example: Video Processing

- A pre-recorded video stream is being decoded and displayed
  - Fetch and decode: get encoded bytes and turn into frames
  - Resize and denoise: transform individual frames before display
  - Display: present the frames to the user
- Consider the metrics: optimise throughput, not execution time
- Video decoding and display have an intrinsic order
- Individual frames can be transformed in parallel in any order

![Diagram](image)

#### Compare and Swap Loop

- **Problem**: a read-write variable is shared between tasks
- **Context**: multiple threads are communicating via some shared variable, such as the best or worst solution seen so far
- **Forces**: the update needs to be thread-safe, but also needs to be fast (so task dependencies cannot be used)
- **Solution**: use an atomic variable combined with a compare and swap loop
- **Examples**: see previous lecture

#### Further design patterns

- There are many more parallel design patterns
  - Some are very specialised for particular applications
  - Some seem extremely obvious
  - People who are into design patterns go a bit over the top
- Best design patterns are those that can be turned into a library
  - Require programmer’s analysis to decide whether it applies
  - Provide a way to quickly apply solution if it is appropriate
- TBB has a number of design patterns as algorithms
  - Section 4 of ref. manual: tbb::parallel_do, tbb::parallel_scan
- Further useful patterns in the TBB Design Patterns Guide
  - *Fenced Data Transfer, Local Serializer, etc.*