The Execution Instance
Overloading Pattern

Douglas Gregor
Andrew Lumsdaine
Open Systems Lab, Indiana University
Motivation: Parallelize a Library

- Start with a sequential library
- Keep the essence of the sequential library
  - Similar syntax
  - Similar semantics
  - Source-code compatibility
- Reuse the sequential implementation
- Efficiency as good as parallel-only library
An introductory example

- **A sequential** `inner_product` **function:**
  ```cpp
double
inner_product(const vec_double& u,
               const vec_double& y);
```

- **Let’s parallelize:**
  - Create a distributed vector type `dist_vec_double`.
  - **Implement distributed** `inner_product`:
    ```cpp
double
inner_product(const dist_vec_double& u,
               const dist_vec_double& y);
```
Execution Instance Overloading

1. Adapt
   - Build a parallel/distributed adaptor
   - Adaptor uses the sequential component
   - Adaptor performs communication and synchronization

2. Overload
   - Give the adaptor the same name as the sequential component
   - Dispatch based on the *execution instance*

3. Optimize
   - Introduce special-case implementations
Execution Instance

- **Execution instance**: The particular context in which a data structure resides or an algorithm is executed.

- **Examples**:
  - Simple, sequential processor machine
  - Vector processor machine
  - Shared-memory multiprocessor machine
  - Cluster of machines communicating via MPI
Old MacDonald had a Queue
E.I.E.I.O.

Efficient Implementation of Execution Instance Overloading
Queue interface

- Queue operations:
  - push(x): add to queue
  - pop(): remove from queue
  - front(): look at front of queue
  - empty(): determine if queue is empty

- Many potential implementations:
  template<typename T>
  class fifo_queue {
    // simple FIFO queue...
    std::deque<T> storage;
  };
Step 1: Adapt

- Build a distributed queue adaptor:

```cpp
template<typename Queue, typename ExecInst>
struct distributed_queue
{
    typedef distributed_value<typename Queue::value_type,
                                ExecInst> value_type;

    void push(value_type x) {ei.send(x.owner(), x.local());}
    void pop() { q.pop();}
    value_type front() { return ei.value(q.front());}
    bool empty() { may synchronize with other processors... }

    Queue q;
    ExecInst ei;
};
```

- Note: Adaptor handles all of the communication!
Step 2: Overload

- “Distributed values” have a specific type: `distributed_value<T, ExecInst>`
- When we see a distributed value type, we need to distribute the queue!
  ```cpp
template<typename T, typename ExecInst>
struct fifo_queue<distributed_value<T, ExecInst> > :
  distributed_queue<fifo_queue<T>, ExecInst>
{
};
```
How overloading works

- User asks for queue<\text{Vertex}>:

  \begin{verbatim}
  template<typename Graph>
  void breadth_first_search(const Graph& g) {
    fifo_queue<typename Graph::Vertex> Q;
    // use the queue for breadth-first search...
  }
  \end{verbatim}

  \begin{itemize}
  \item If \text{Graph} is not distributed, we get a non-distributed queue
  \item If \text{Graph} is distributed, its \text{Vertex} type is a \text{distributed\_value}, and we get a distributed queue.
  \end{itemize}

- The execution instance of the graph “flows” to the queue.
Step 3: Optimize

- Adaptor implementations can be too naïve
- Consider a `locking_queue` adaptor
  - Okay with little contention, complex queues
  - Too slow with FIFO queue and high contention!
- A “real” parallel library might have used a lockless queue.
Overloading for optimization

- Lockless queues can be used when
  - The user requested a FIFO queue
  - We’re operating in shared memory
  - Compare-and-swap exists

- Implement this as:
  ```cpp
template<typename T>
struct locking_queue<fifo_queue<T>, cas_exec_instance>
  : lockless_fifo_queue<T>
{
};
```
The Parallel BGL

- Parallelizes the Boost Graph Library (BGL)
  - The BGL interface is retained
  - Implemented on top of the BGL
  - Supports various execution instances
- Makes heavy use of the Execution Instance Overloading pattern
Distributed Adjacency List

- Adaptor over BGL adjacency list
  - Divide vertices among processors
  - Each processor stores the subgraph of edges originating from its own vertices
  - Adaptor handles all communication
Distributed Dijkstra’s Algorithm

- Overloads sequential Dijkstra’s algorithm
  - Mimics the interface
  - Client code need not change
- Implemented as a call to BGL
  `breadth_first_search()`
  - Graph, property maps, queue are distributed
  - Queue uses lookahead heuristics to improve performance
Why C++ templates?
Some abstractions hurt
Parallel BGL: Fixed Problem Size
Parallel BGL: Scaled Problem Size

- Breadth-First Search
- Crauser et al. Dijkstra
- Eager Dijkstra 0.1
- Connected Components
- Boman et al Coloring
Summary

- EIO pattern is for building parallel libraries from sequential libraries
  - Adapt
  - Overload
  - Optimize
- Efficient Implementation of EIO pattern via C++ templates
- Parallel BGL uses EIO to good effect. See: http://www.osl.iu.edu/research/pbgl