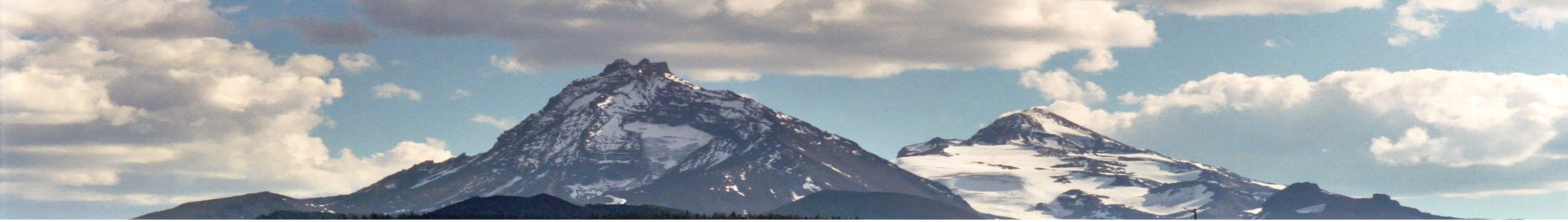


Pipeline Pattern

Parallel Computing

CIS 410/510

Department of Computer and Information Science

UNIVERSITY OF OREGON

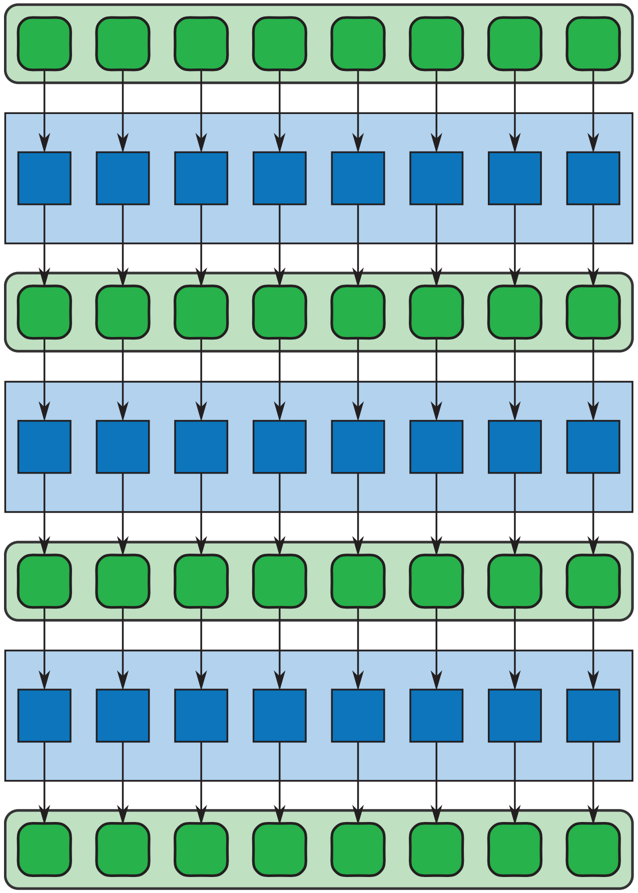
Table of Contents

- Concept
  - Example
- Implementation
- Tool Support
  - TBB
  - Cilk Plus
- Example

CONCEPT

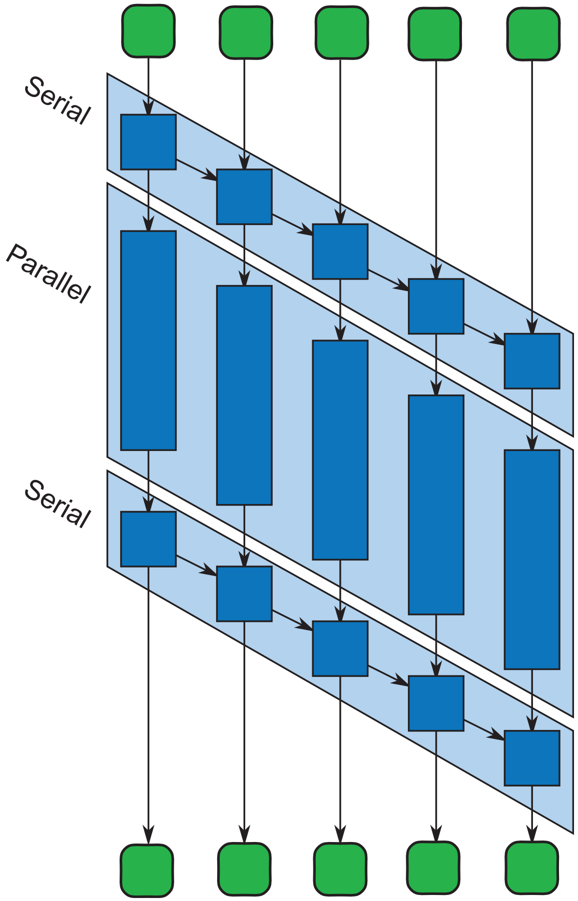
Combining Maps

- Map operations are often performed in sequence
- Can sometimes optimize this as one big map
- Not always feasible
  - Steps may be in different modules or libraries
  - There may be serial operations interleaved



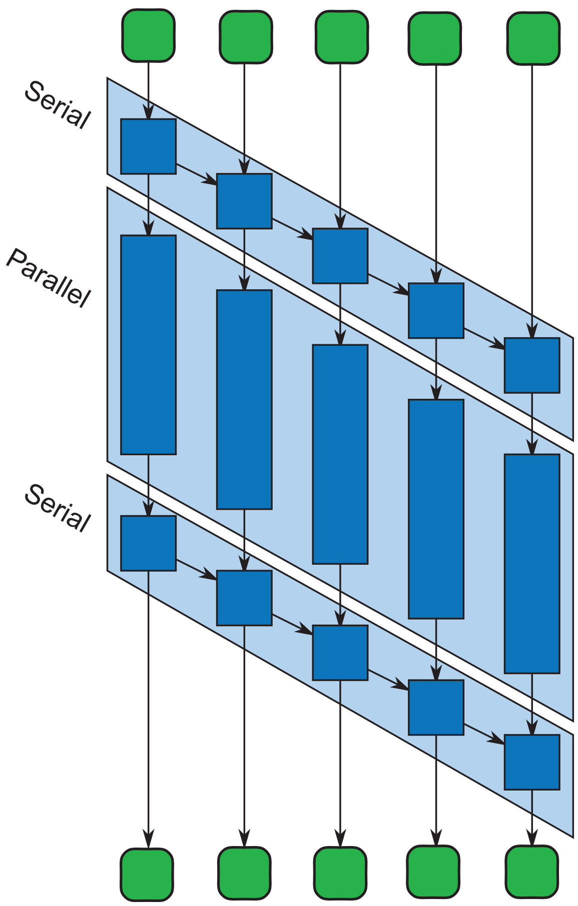
The Pipeline Pattern

- A *pipeline* is composed of several computations called *stages*
  - Parallel stages run independently for each item
  - Serial stages must wait for each item in turn



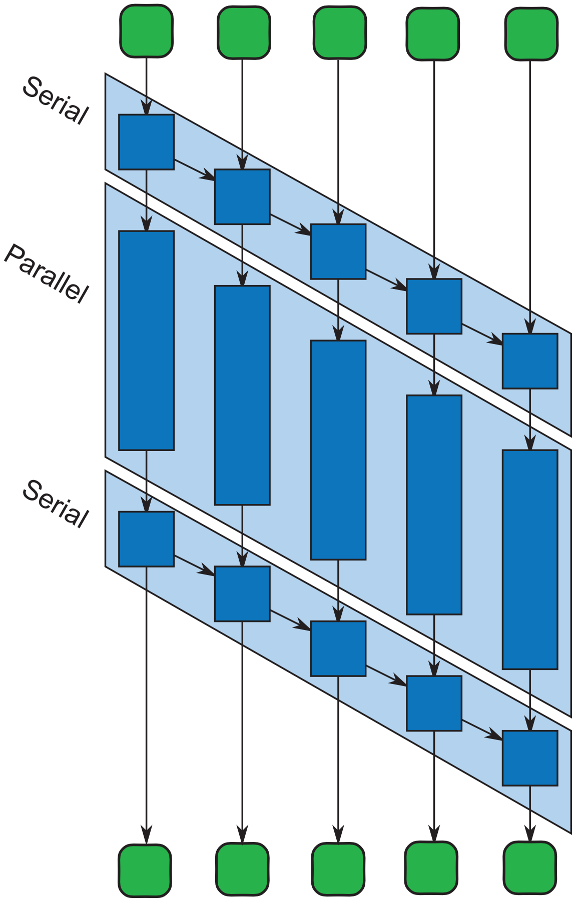
The Pipeline Pattern

- Advantages:
  - Conceptually simple
  - Allows for modularity
  - Parallelizes as much as possible, even when some stages are serial, by overlapping
  - Accommodates I/O as serial stages



The Pipeline Pattern

- Disadvantages:
  - Serial computation is still a bottleneck
  - Somewhat difficult to implement well from scratch



Example: Bzip2 Data Compression

- The `bzip2` utility provides general-purpose data compression
  - Better compression than `gzip`, but slower
- The algorithm operates in blocks
  - Blocks are compressed independently
  - Some pre- and post-processing must be done serially

Three-Stage Pipeline for Bzip2

- Input (serial)
  - Read from disk
  - Perform run-length encoding
  - Divide into blocks
- Compression (parallel)
  - Compress each block independently
- Output (serial)
  - Concatenate the blocks at bit boundaries
  - Compute CRC
  - Write to disk

IMPLEMENTATION

Implementation Strategies

- Stage-bound workers
  - Each stage has a number of workers
    - Serial stages have only one
  - Each worker takes a waiting item, performs work, then passes item to the next stage
  - Essentially the same as map
  - Simple, but no data locality for each item

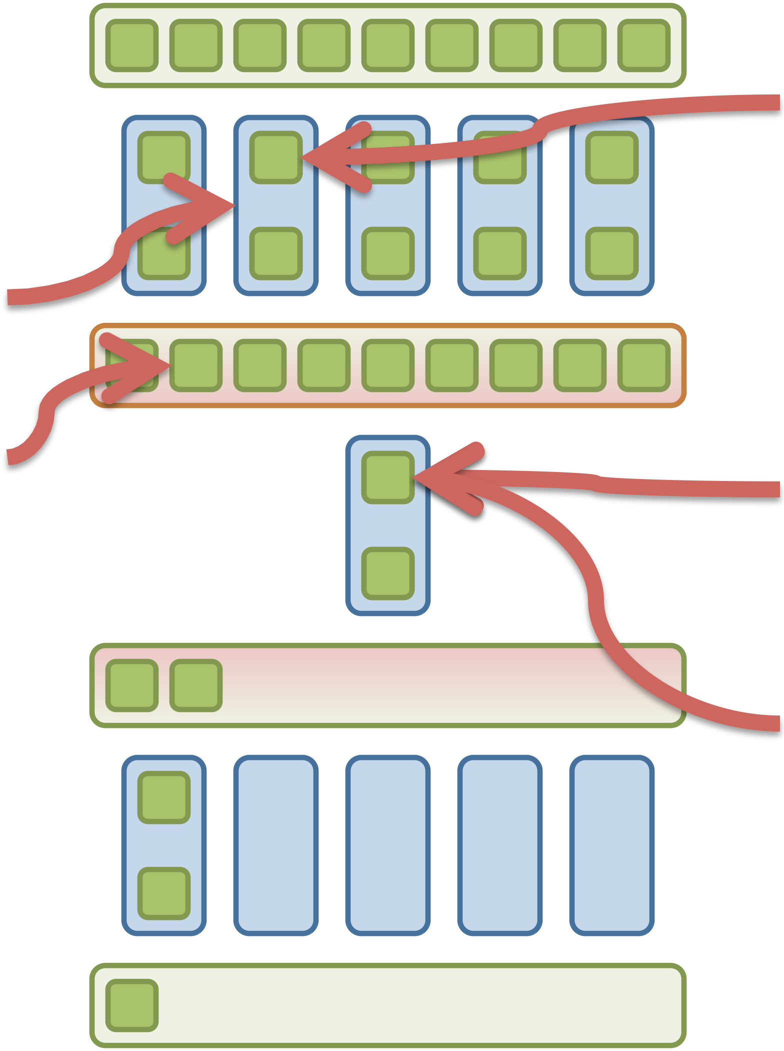
Stage-Bound Workers

First, each worker grabs input and begins processing it.

Suppose this one finishes first.

The item gets passed to the serial stage.

Since it's out of order, it must wait to be processed.



Meanwhile, the finished worker grabs more input.

The serial stage accepts the first item.

Now that the first item is processed, the second one can enter the serial stage.

Implementation Strategies

- Item-bound workers
  - Each worker handles an item at a time
  - Worker is responsible for item through whole pipeline
  - On finishing last stage, loops back to beginning for next item
  - More complex, but has much better data locality for items
    - Each item has a better chance of remaining in cache throughout pipeline
  - Workers can get stuck waiting at serial stages

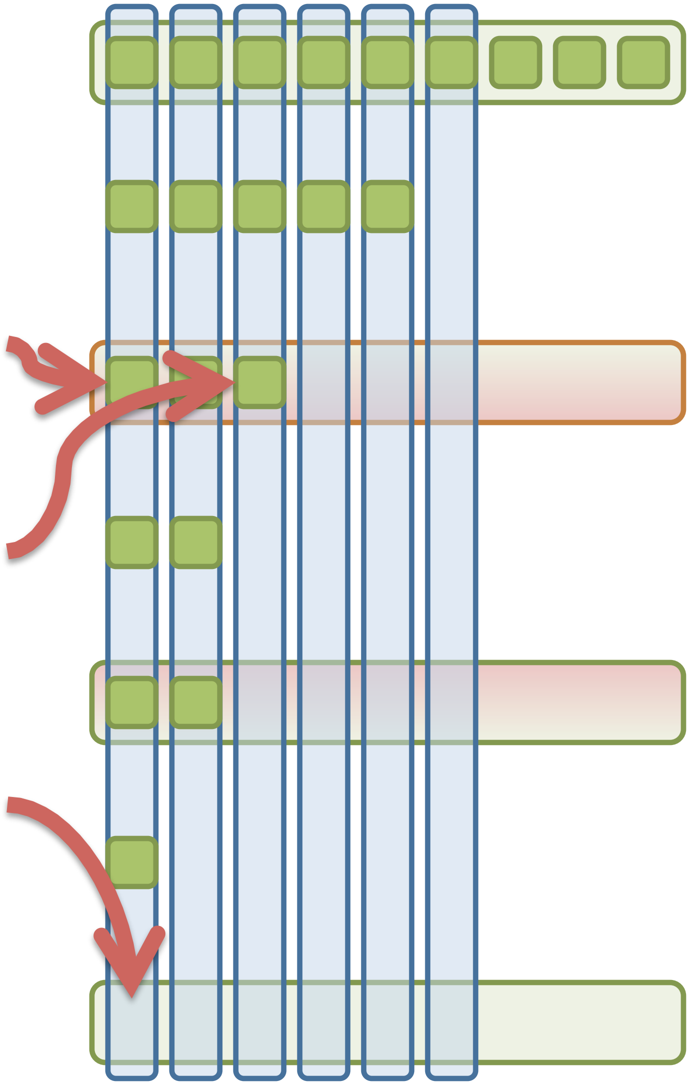
Item-Bound Workers

Each worker gets an item, which it carries through the pipeline.

If an item arrives at a serial stage in order, the worker continues.

Otherwise, it must block until its turn comes.

When an item reaches the end, its worker starts over at the first stage.



Implementation Strategies

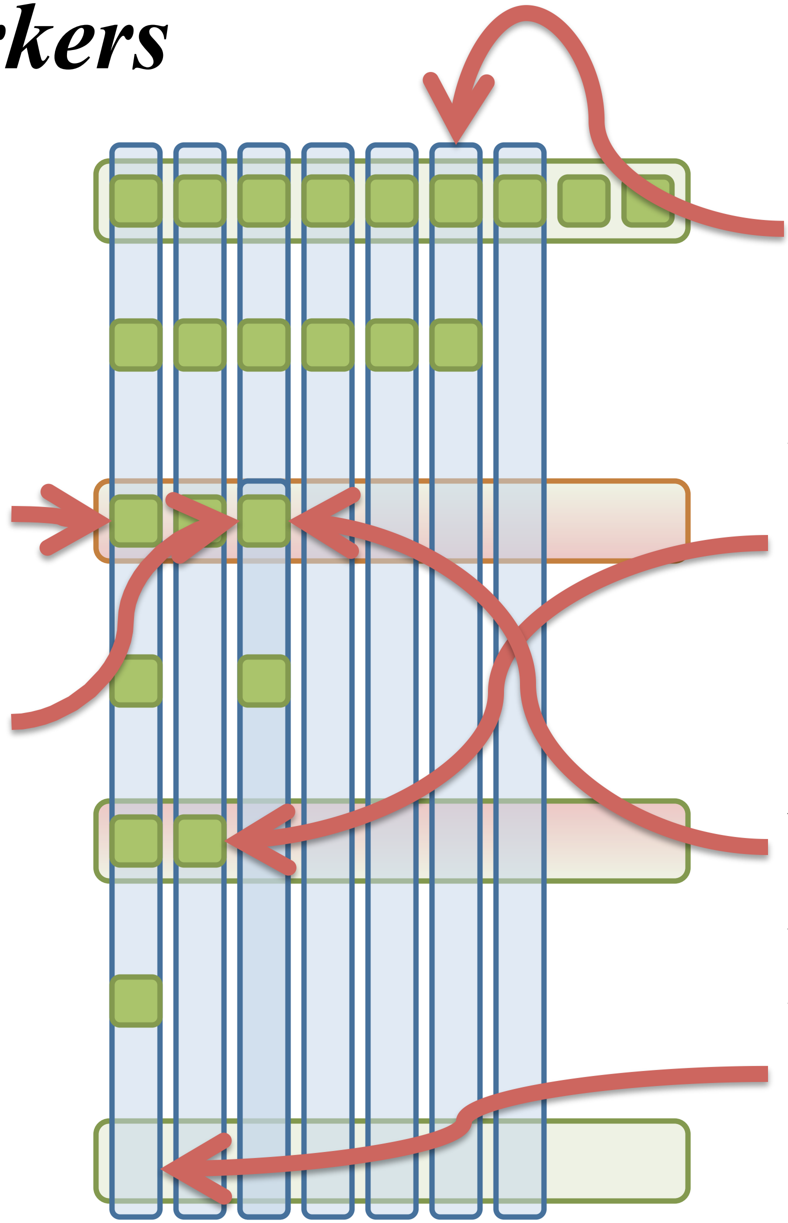
- Hybrid (as implemented in TBB)
  - Workers begin as item-bound
  - When entering a serial stage, the worker checks whether it's ready to process the item now
    - If so, the worker continues into the stage
    - Otherwise, it *parks* the item, leaving it for another worker, and starts over
  - When leaving a serial stage, the worker checks for a parked item, spawning a new worker to handle it
  - Retains good data locality without requiring workers to block at serial stages
  - No locks needed; works with greedy schedulers

Hybrid Workers

Each worker gets an item, which it intends to carry through the pipeline.

If an item arrives at a serial stage in order, its worker continues.

Otherwise, the worker "parks" the item and abandons it …



… starting over at the first stage.

Whenever an item finishes a serial stage, it checks for a parked item.

If there is one, a new worker is spawned to go through the rest of the pipeline.

When a worker finishes, it starts over at the first stage.

TOOL SUPPORT

Pipelines in TBB

- Built-in support from the `parallel_pipeline` function and the `filter_t` class template
- A `filter_t<X, Y>` takes in type `X` and produces `Y`
  - May be either a serial stage or a parallel stage
- A `filter_t<X, Y>` and a `filter_t<Y, Z>` combine to form a `filter_t<X, Z>`
- `parallel_pipeline()` executes a `filter_t<void, void>`

Pipelines in TBB: Example Code

- This is a three-stage pipeline with serial stages at the ends and a parallel stage in the middle.
- Here, `f` is a function that returns successive items of type `T` when called, eventually returning `NULL` when done
  - Might not be thread-safe
- `g` comprises the middle stage, mapping each item of type `T` to one item of type `U`
  - Must be thread-safe
- `h` receives items of type `U`, in order
  - Might not be thread-safe

```
void tbb_sps_pipeline( size_t ntoken ) {
   tbb::parallel_pipeline (
      ntoken,
      tbb::make_filter<void,T>(
         tbb::filter::serial_in_order,
         [&]( tbb::flow_control& fc ) -> T{
            T item = f();
            if( !item ) fc.stop();
            return item;
         }
      ) &
      tbb::make_filter<T,U>(
         tbb::filter::parallel,
         g
      ) &
      tbb::make_filter<U,void>(
         tbb::filter::serial_in_order,
         h
      )
   );
}
```

Pipelines in TBB: Example Code

- Note the `ntoken` parameter to `parallel_pipeline`
  - Sets a cap on the number of items that can be in processing at once
  - Keeps parked items from accumulating to where they eat up too much memory
  - Space is now bound by `ntoken` times the space used by serial execution

```
void tbb_sps_pipeline( size_t ntoken ) {
   tbb::parallel_pipeline (
      ntoken,
      tbb::make_filter<void,T>(
         tbb::filter::serial_in_order,
         [&]( tbb::flow_control& fc ) -> T{
            T item = f();
            if( !item ) fc.stop();
            return item;
         }
      ) &
      tbb::make_filter<T,U>(
         tbb::filter::parallel,
         g
      ) &
      tbb::make_filter<U,void>(
         tbb::filter::serial_in_order,
         h
      )
   );
}
```

Pipelines in Cilk Plus

- No built-in support for pipelines
- Implementing by hand can be tricky
  - Can easily fork to move from a serial stage to a parallel stage
  - But can’t simply join to go from parallel back to serial, since workers must proceed in the correct order
  - Could gather results from parallel stage in one big list, but this reduces parallelism and may take too much space

Pipelines in Cilk Plus

- Idea: A reducer can store sub-lists of the results, combining adjacent ones when possible
  - By itself, this would only implement the one-big-list concept
  - However, whichever sub-list is farthest left can process items immediately
    - The list may not even be stored as such; can "add" items to it simply by processing them
  - This way, the serial stage is running as much as possible
  - Eventually, the leftmost sub-list comprises all items, and thus they are all processed

Pipelines in Cilk Plus: Monoids

- Each view in the reducer has a sub-list and an `is_leftmost` flag
- The views are then elements of two monoids*
  - The usual list-concatenation monoid (a.k.a. the *free monoid*), storing the items
  - A monoid* over Booleans that maps $x \otimes y$ to x, keeping track of which sub-list is leftmost
    - *Not *quite* actually a monoid, since a monoid has to have an identity element I for which $I \otimes y$ is always y
    - But close enough for our purposes, since the only case that would break is `false` $\otimes$ `true`, and the leftmost view can't be on the right!
- Combining two views then means concatenating adjacent sub-lists and taking the left `is_leftmost`

Pipelines in Cilk Plus: Example Code

- Thus we can implement a serial stage following a parallel stage by using a reducer to mediate them
- We call this a *consumer reducer*, calling the class template `reducer_consume`

```
#include <cilk/reducer.h>
#include <list>
#include <cassert>

template<typename State, typename Item>
class reducer_consume {
public:
   // Function that consumes an Item to update a State object
   typedef void (*func_type)(State*,Item);
private:
   struct View {
      std::list<Item> items;
      bool is_leftmost;
      View( bool leftmost=false ) : is_leftmost(leftmost) {}
      ~View() {}
   };

   struct Monoid: cilk::monoid_base<View> {
      State* state;
      func_type func;
      void munch( const Item& item ) const {
         func(state,item);
      }
      void reduce(View* left, View* right) const {
         assert( !right->is_leftmost );
         if( left->is_leftmost )
            while( !right->items.empty() ) {
               munch(right->items.front());
               right->items.pop_front();
            }
         else
            left->items.splice( left->items.end(), right->items );
      }
      Monoid( State* s, func_type f ) : state(s), func(f) {}
   };
```

Pipelines in Cilk Plus: Example Code

- A `View` instance is an element of the (not-quite-)monoid.

```
#include <cilk/reducer.h>
#include <list>
#include <cassert>

template<typename State, typename Item>
class reducer_consume {
public:
   // Function that consumes an Item to update a State object
   typedef void (*func_type)(State*,Item);
private:
   struct View {
      std::list<Item> items;
      bool is_leftmost;
      View( bool leftmost=false ) : is_leftmost(leftmost) {}
      ~View() {}
   };

   struct Monoid: cilk::monoid_base<View> {
      State* state;
      func_type func;
      void munch( const Item& item ) const {
         func(state,item);
      }
      void reduce(View* left, View* right) const {
         assert( !right->is_leftmost );
         if( left->is_leftmost )
            while( !right->items.empty() ) {
               munch(right->items.front());
               right->items.pop_front();
            }
         else
            left->items.splice( left->items.end(), right->items );
      }
      Monoid( State* s, func_type f ) : state(s), func(f) {}
   };
```

Pipelines in Cilk Plus: Example Code

- The `Monoid` class implements the `reduce` function.
  - The leftmost sub-list is always empty; rather than add items to it, we process all of them immediately
- The `func` field holds the function implementing the serial stage
- The `state` field is always passed to `func` so that the serial stage can be stateful

```
#include <cilk/reducer.h>
#include <list>
#include <cassert>

template<typename State, typename Item>
class reducer_consume {
public:
   // Function that consumes an Item to update a State object
   typedef void (*func_type)(State*,Item);
private:
   struct View {
      std::list<Item> items;
      bool is_leftmost;
      View( bool leftmost=false ) : is_leftmost(leftmost) {}
      ~View() {}
   };

   struct Monoid: cilk::monoid_base<View> {
      State* state;
      func_type func;
      void munch( const Item& item ) const {
         func(state,item);
      }
      void reduce(View* left, View* right) const {
         assert( !right->is_leftmost );
         if( left->is_leftmost )
            while( !right->items.empty() ) {
               munch(right->items.front());
               right->items.pop_front();
            }
         else
            left->items.splice( left->items.end(), right->items );
      }
      Monoid( State* s, func_type f ) : state(s), func(f) {}
   };
```

Pipelines in Cilk Plus: Example Code

- To use the consumer reducer, the parallel stage should finish by invoking consume with the item
- If this worker has the leftmost view, the item will be processed immediately; otherwise it is stored for later
  - Similar to the hybrid approach, but with less control
- Following a `cilk_sync`, all items will have been processed
  - Including the implicit sync at the end of a function or `cilk_for` loop

```

    cilk::reducer<Monoid> impl;

public:
    reducer_consume( State* s, func_type f ) :
        impl(Monoid(s,f), /*leftmost=*/true)
    {}

    void consume( const Item& item ) {
        View& v = impl.view();
        if( v.is_leftmost )
            impl.monoid().munch( item );
        else
            v.items.push_back(item);
    }
};
```

EXAMPLE: BZIP2 COMPRESSION

Parallel Bzip2 in Cilk Plus

- The `while` loop comprises the first stage
  - Reads in the file, one block at a time, spawning a call to `SecondStage` for each block
- `SecondStage` compresses its block, then passes it to the consumer reducer
  - Reducer preconfigured (line 17) to invoke `ThirdStage`
- `ThirdStage` always receives the blocks in order, so it outputs blocks as it receives them

```
1  void SecondStage( EState* s, reducer_consume<OutputState, EState*>& sink ) {
2     if( s->nblock )
3        CompressOneBlock(s);
4     sink.consume( s );
5  }
6
7  void ThirdStage( OutputState* out_state, EState* s ) {
8     if( s->nblock )
9        out_state->putOneBlock(s);
10    FreeEState(s);
11 }
12
13 int BZ2_compressFile(FILE *stream, FILE *zStream, int
       blockSize100k, int verbosity, int workFactor) throw()
       {
14    ...
15    InputState in_state;
16    OutputState out_state( zStream );
17    reducer_consume<OutputState,EState*> sink(&out_state, ThirdStage);
18    while( !feof(stream) && !ferror(stream) ) {
19       EState *s = BZ2_bzCompressInit(blockSize100k, verbosity, workFactor);
20       in_state.getOneBlock(stream,s);
21       cilk_spawn SecondStage(s, sink);
22    };
23    cilk_sync;
24    ...
25 }
```