Lecture 9 – Fork-Join Pattern

CIS 410/510
Parallel Computing
Pipeline Pattern

Prof. Allen D. Malony
Department of Computer and Information Science
Spring 2016

UNIVERSITY OF OREGON
Logistics

- Homework
  - Homework #2 posted
    - Due Tuesday, May 3, 5pm
- Term projects
  - Decided to have a class project
  - Will start discussion at the end of class
- Term papers (graduate students)
- OpenACC is the programming lab this week
Contents

- What is the pipeline concept?
- What is the pipeline pattern?
- Pipeline performance
- Example: Bzip2 data compression
- Implementation strategies
- Pipelines in TBB
- Pipelines in Cilk Plus
- Example: parallel Bzip2 data compression
- Mandatory parallelism vs. optional parallelism
Pipeline

- A pipeline is a linear sequence of stages
- Data flows through the pipeline
  - From Stage 1 to the last stage
  - Each stage performs some task
    - uses the result from the previous stage
  - Data is thought of as being composed of units (items)
  - Each data unit can be processed separately in pipeline
  - Sequence of stages (tasks) matters (functional operation)
  - Data sequence might matter
- Pipeline computation is a special form of producer-consumer parallelism
  - Producer tasks output data …
  - … used as input by consumer tasks
**Pipeline Model**

- **Stream** of data operated on by succession of tasks
  - Assumption: data input and output must be in sequence

- Each task is done in a separate stage

- Consider 3 data units and 4 tasks (stages)
  - Sequential pipeline execution (no parallel execution)
Where is the Concurrency? (Serial Pipeline)

- Pipeline with serial stages
  - Each stage runs serially (i.e., can not be done in parallel)
  - Assume that we can not parallelize the tasks (for now)

- What can we run in parallel?
  - Think about data parallelism
  - Provide a separate pipeline for each data item

- What do you notice as we increase # data items?
Where is the Concurrency? (Serial Pipeline)

What is happening here in this region?

10 data items

How much parallelism is there?

startup

Finish

Processor
1
2
3
4
5
6
7
8
9
10

End

finish
Parallelizing Serial Pipelines

- # tasks limits the parallelism with serial tasks
- Data sequence limits parallelism
- Two parallel execution choices:
  1) processor executes the entire pipeline
  2) processor assigned to execute a single task
- Which is better? Why?
Parallelizing Serial Pipeline

Can we do better with 5 processors? Why or why not?
Lecture 9 – Fork-Join Pattern

Pipeline Performance

- N data and T tasks
- Each task takes unit time t
- Sequential time = N*T*t = NT (t=1)
- Parallel pipeline time
  = start + finish + parallel
  = T-1 + T-1 + (N-2(T-1))/T
  = 2T-2 + N/T + (2T-2)/T = N/T + (2T-2)(1+1/T)
  = O(N/T) (for N>>T)

- Try to find a lot of data to pipeline
- Try to divide computation in a lot of pipeline tasks
  - More tasks to do (longer pipelines) = more parallelism
  - Shorter tasks to do (as a result of breaking apart tasks)
- Interested in pipeline throughput
Pipeline Performance

- $N$ data and $T$ tasks
- Suppose the tasks execution times are non-uniform
- Suppose a processor is assigned to execute a task
- What happens to the throughput?
- What limits performance?
- Slowest stage limits throughput … Why?
- Little’s Law comes into play
Basic Throughput Quantities

- At all levels of the system (register files through networks), there are three fundamental (efficiency-oriented) quantities:
  - **Latency**
    - every operation requires time to execute
    - # stages (tasks) in a pipeline * time for each task
  - **Bandwidth**
    - # of (parallel) operations completed per cycle
  - **Concurrency**
    - total # of operations in flight
**Little’s Law**

- In queueing theory, Little’s Law expresses a relationship between latency and throughput in a stable system (e.g., a pipeline with infinite data)

\[
\text{occupancy (concurrency)} = \text{latency} \times \text{throughput}
\]

- Think of a water pipe
  - Throughput = rate at which water is put into the pipe
  - Latency = “length” of the pipe (processing time for data)

- If increase throughput, need more concurrency
  - How do you get it? … increase # pipeline stages

- If latency decreases, need more concurrency
**Pipeline Model with Parallel Stages**

- What if we can parallelize a task?
- What is the benefit of making a task run faster?
- Book describes 3 kinds of stages (with Intel TBB):
  - Parallel: processing incoming items in parallel
  - Serial out of order: process items 1 at a time (arbitrary)
  - Serial in order: process items 1 at a time (in order)
- Tasks are ordered in a pipeline, but data might not necessarily be ordered
- What advantages might there be if the pipeline outputs do not have to be the same as order as the inputs?
Serial-Parallel-Serial Pipeline Pattern

- Simplest common sequence of stages for a parallel pipeline
- Serial stages are in order
- Feedback in serial stage indicates that data items are processes in order
- Lack of feedback in parallel stage means data items can be processed in parallel
Parallel 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
- Input order is the same as the output order
- Stages are ordered for each data element
**Parallel 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
Parallel Pipeline Pattern

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

• What are the reasons for serial stages?
  – Serial input?
    • determined mainly by data availability on input and/or how data input is processes
  – Serial output?
    • similar reasons
Combining Maps

- Map operations are often performed in sequence
- Can think of this as a pipeline of maps
- 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
Example: Bzip2 Data Compression

- **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 (cyclic redundancy checksum)
  - Write to disk
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
  - No benefit of 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 … Why?
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.
**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: SPS Example

- 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

```cpp
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
    )
  );
}
```

LISTING 9.2
Pipeline in TBB. It is equivalent to the serial pipeline in Listing 9.1, except that the stages run in parallel and the middle stage processes multiple items in parallel.

The first and last stages use side effects to communicate with the rest of the program, and the corresponding input/output types are `void`. The last stage is declared as mapping items from `U` to `void` and uses side effects to output items. Conversely, the first stage is declared as mapping items from `void` to `T` and uses side effects to input items.

The first stage is also special because each time it is invoked it has to return an item or indicate that there are no more items. The TBB convention is that it takes an argument of type `flow_control&`. If it has no more items to output, it calls method `stop` on that argument, which indicates that there are no more items, and the currently returned item should be ignored.
Pipelines in TBB: SPS Example

- 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
    )
  );
}
```

Listing 9.2

Pipeline in TBB. It is equivalent to the serial pipeline in Listing 9.1, except that the stages run in parallel and the middle stage processes multiple items in parallel.

The first and last stages use side effects to communicate with the rest of the program, and the corresponding input/output types are `void`. The last stage is declared as mapping items from `U` to `void` and uses side effects to output items. Conversely, the first stage is declared as mapping items from `void` to `T` and uses side effects to input items.

The first stage is also special because each time it is invoked it has to return an item or indicate that there are no more items. The TBB convention is that it takes an argument of type `flow_control&`. If it has no more items to output, it calls method `stop` on that argument, which indicates that there are no more items, and the currently returned item should be ignored.
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: Possible Approach

- 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 $\text{false} \otimes \text{true}$, 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: SPS Example

• 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`.

Listing 9.4 shows the implementation. The list is non-empty only if actual parallelism occurs, since only then is there a non-leftmost view.

```
#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) {}  //View() {} 
    };
```
Pipelines in Cilk Plus: SPS Example

- A View instance is an element of the (not-quite-)monoid
Pipelines in Cilk Plus: SPS Example

- 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

```cpp
#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: SPS Example

- 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.

```cpp
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);
    }
```

From a mathematical perspective, the fields of `View` are monoid values:
- `View::items` is a value in a list concatenation monoid.
- `View::is_leftmost` is a value in a monoid over boolean values, with operation $x \lor y = x$. Both of these operations are associative but not commutative.

9.5 MORE GENERAL TOPOLOGIES

Pipelines can be generalized to non-linear topologies, and stages with more complex rules. TBB 4.0 has such a framework, in namespace `tbb::flow`. This framework also lifts the restriction that each stage map exactly one input item to one output item. With more complex topologies comes power, but more programmer responsibility.

The additional power of the TBB 4.0 `tbb::flow` framework comes with additional responsibility. The framework lifts most of the restrictions of `parallel_pipeline` while still using a modified `bound-to-item` approach that avoids explicit waiting. It allows stages to perform one-to-many and many-to-one mappings, and topologies can be non-linear and contain cycles. Consequently, designing pipelines in that framework requires more attention to potential deadlock if cycles or bounded buffers are involved.

9.6 MANDATORY VERSUS OPTIONAL PARALLELISM

Different potential implementations of a pipeline illustrate the difference between optional parallelism and mandatory parallelism. Consider a two-stage pipeline where the producer puts items into a bounded buffer and the consumer takes them out of the buffer. There is no problem if the producer
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

```cilk
void SecondStage( EState* s, reducer_consume<OutputState, EState*>& sink ) {
    if ( s->nblock )
        CompressOneBlock(s);
    sink.consume( s );
}

void ThirdStage( OutputState* out_state, EState* s ) {
    if( s->nblock )
        out_state->putOneBlock(s);
    FreeEState(s);
}

int BZ2_compressFile(FILE *stream, FILE *zStream, int blockSize100k, int verbosity, int workFactor) throw() {

    InputState in_state;
    OutputState out_state( zStream );
    reducer_consume<OutputState,EState*> sink(&out_state, ThirdStage);
    while( !feof(stream) && !ferror(stream) ) {
        EState *s = BZ2_bzCompressInit(blockSize100k, verbosity, workFactor);
        in_state.getOneBlock(stream,s);
        cilk_spawn SecondStage(s, sink);
    }
    cilk_sync;

    /* file passed to BZ2_compressInit(); */
    cilk_barrier;

    return 0;
}
```
Mandatory vs. Optional Parallelism

- Consider a 2-stage (producer-consumer) pipeline
  - Produces puts items into a buffer for consumer
- There is no problem if the producer and consumer run in parallel
- The serial execution is tricky because buffer can fill up and block progress of the producer
  - Similar situation with the consumer
  - Producer and consumer must interleave to guarantee progress
- Restricting the kinds of pipelines that can be built
  - No explicit waiting because a stage invoked only when its input item is ready and must emit exactly 1 output
  - Going from parallel to serial require buffering
- Mandatory parallelism forces the system to execute operations in parallel whereas optional does not require it