CHAPTER

12 Bzip2 Data Compression

Bzip2 is a popular file compression program. We show it as a practical application of the **pipeline** pattern. The pattern is more than a classic pipeline because for the sake of scalability some stages process multiple items simultaneously. The TBB implementation shows `parallel_pipeline` in action. The Cilk Plus implementation demonstrates how creative use of **reducers** can go far beyond implementing mathematically pure **reductions**. Another point of the example is that, when designing file formats, you should consider their impact on parallel processing.

12.1 THE BZIP2 ALGORITHM

Bzip2 chops its input stream into blocks, compresses each block separately, and writes the blocks to an output stream. Two features of the output stream affect parallelization considerations:

- Each output block starts on the bit (not byte!) immediately following the previous block.
- A cyclic redundancy check (CRC) code is written after all blocks are written.

The overall steps of the algorithm are:

```
while( not at end of the input stream ) {
    Read a block from the input stream
    Compress the block
    Update CRC
    Realign the block on a bit boundary
    Write the block to the output stream
}
Output final CRC
```

In practice, the realignment and writing can be optimized by fusing them into a single operation that realigns and writes a block, instead of realigning the entire block before writing any of it. Better yet, to reduce the memory footprint of these two fused steps, the block can be treated as a sequence of chunks, and each chunk can be aligned/written separately.

Compression of a block involves the following operations:

1. Run-length encoding, which cheaply compresses long runs of the same character.
2. Burrows–Wheeler transform [BW94], which permutes the block in a way that tends to put similar substrings close to each other. The transform sorts all possible cyclic rotations of the block and, using the sorted order, outputs the *last* letter of each rotation. It also outputs the index of where

Structured Parallel Programming. DOI: 10.1016/B978-0-12-415993-8.00012-8

the original block occurs in the sorted list of rotations. The original block can be reconstructed from this information [BW94]. For an example of the forward transform, let the block be the string `LOGHOG`. The result of the sort is:

Index	Rotations	Sorted Rotations
0	LOGHOG	GHOGLO
1	OGHOGL	GLOGHO
2	GHOGLO	HOGLOG
3	HOGLOG	LOGHOG
4	OGLOGH	OGHOGL
5	GLOGHO	OGLOGH

The output of the transform is the last letter of each entry in the sorted list: `OOGGLH`, and the index 3, which is the index of `LOGHOG` in the sorted list. In practice, the block can be as long as 900 kilobytes, so the sort can take significant time.

3. Move-to-front coding [BSTW86], which transforms the block so similar substrings that are close to each other produce the same output symbol.
4. Another round of run-length encoding.
5. Huffman coding, which compresses the transformed block, which at this point tends to have high redundancy of some symbols.

In principle, the run-length encoding can be parallelized by parallel **scan**, the Burrows–Wheeler transform can use a parallel sort, and the Huffman transform is a parallelizable **map** pattern. However, the move-to-front transform is serial. Hence, it is more practical to exploit parallelism across blocks, not inside blocks.

Another consideration in favor of parallelizing across blocks is that reading and writing blocks contribute a significant portion of the work. Hence, for good speedup it is critical to do reading, compressing, and writing **concurrently**.

12.2 THREE-STAGE PIPELINE USING TBB

Figure 12.1 sketches a three-stage **pipeline** implementation that can be implemented using `tbb::parallel_pipline`. The first and last stages are serial. They each process one block at a time. The `tbb::parallel_pipline` template guarantees that each serial stage processes items in the same order as the previous serial stage. The middle stage is parallel and does most of the compression work.

However, the initial run-length encoding is done in the serial stage. This is to maintain bitwise compatibility with serial bzip2. In serial bzip2, the blocksize is the maximum size of a block *after* the initial run-length encoding. So block boundaries are determined after run-length encoding. As remarked earlier, parallel scan could be used inside the first stage to parallelize the encoding, though given the low **arithmetic intensity** of run-length encoding it is unlikely to pay off anyway.

Bzip2 is a large application, so only a tiny fraction of the code is shown, specifically the fraction needed to understand how it was parallelized using `tbb::parallel_pipeline`. In the serial

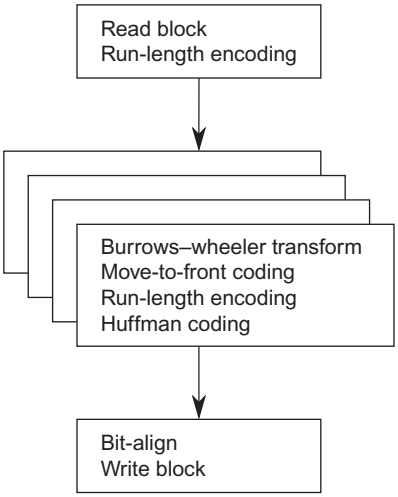

FIGURE 12.1

Three-stage pipeline for bzip2.

version of bzip2, a data structure of type `EState` describes the current block being compressed. It has many fields. For our purposes here, only the fields related to the initial run-length encoding need to be understood.

```
struct EState {
    ...
    UInt32 state_in_ch;   // Most recently read byte
    Int32  state_in_len;  // Current run-length of state_in_char
    ...
    Int32  nblock;        // Number of input bytes in block
    ...
};
```

In the serial version, there is a single instance of `EState`. The parallel version needs to be reading the next block, and compressing multiple blocks in parallel, so the parallel version needs one instance of `EState` per block in flight. However, the two fields shown carry information between blocks. One solution would be to move them to a different structure associated with only the input stage. However, for the sake of minimizing changes to the serial code, a slightly different approach is employed. After the initial pipeline stage reads a block, it remembers the two field values from the current `EState`, and uses those values to initialize the `EState` instance for the next block.
Listing 12.1 declares the state for the input stage and the routine that gets a block.

The output stage has some similar information-carrying issues.

- Accumulate the total volume written.
- Create a a CRC that is the combination of the CRC values of each block.
- Concatenate compressed blocks on bit boundaries. When writing a block, any bits beyond the most recent byte written must be included in the first byte written for the subsequence block.

```
class InputState {
public:
    InputState() : prev_state_char(256), prev_state_len(0), nbytes_in(0),
        buf_ptr(NULL), buf_end(NULL) {}
    EState* getOneBlock(FILE* inputFile, EState* s);
    // Volume of uncompressed data
    off_t nbytes_in;
private:
    // Values carried between instances of EState
    UInt32 prev_state_char;
    Int32 prev_state_len;
    // Field and routine related to buffering input
    UChar *buf_ptr;
    UChar *buf_end;
    UChar buf[8192];
    void copy_input_until_stop( EState* s, FILE* infile );
};

void InputState::getOneBlock(FILE* inputFile, EState* s) {
    // Carry information from previous EState
    s->state_in_ch = prev_state_char;
    s->state_in_len = prev_state_len;

    copy_input_until_stop( s, inputFile );

    // Remember information to be carried to next EState
    prev_state_char = s->state_in_ch;
    prev_state_len = s->state_in_len;

    // Accumulate total input volume
    nbytes_in += s->nblock;
}
```

LISTING 12.1

Declarations for bzip2 pipeline.

This state is declared in a class `OutputState`. For brevity, a listing is not provided here, but it is analogous to the way `InputState` was done.

The middle compression stage carries no information between blocks, so it needs no corresponding state object. It just calls a function on the the `Estate` object passing through the stage.

Now the TBB syntactic mechanics for building and running the pipeline can be detailed. The routine `tbb::parallel_pipeline` takes two parameters:

- A token limit, which is an upper bound on the number of items that are processed simultaneously. Without this limit, a traffic jam behind a serial stage could pile up forever, eventually exhausting memory. Here, a limit equal to the number of hardware threads is reasonable.

- A pipeline constructed by composing `filter_t` objects together with `&`. Each object is created by function `tbb::make_filter`.

Listing 12.2 shows the key parts of the code. The template arguments to `make_filter` indicate the type of input and output items for the filter. The ordinary arguments specify whether the `filter_t` is parallel or not and specify a functor that maps an input item to an output item. For example, the middle stage is parallel and maps a `Estate*` to an `Estate*` using `CompressOneBlock`. The last stage has an output type of `void` since it is consuming items, not mapping them.

The functor in the first stage is a special case since it is producing items, not mapping items to items. The argument to it is not an item, but a special object of type `tbb::flow_control` used to

```
1  int BZ2_compressFile(FILE *stream, FILE *zStream, int
       blockSize100k, int verbosity, int workFactor) throw() {
2      …
3      InputState in_state;
4      OutputState out_state( zStream );
5      tbb::parallel_pipeline(
6         ntoken,
7         tbb::make_filter<void,EState*>( tbb::filter::
              serial_in_order, [&]( tbb::flow_control& fc ) -> EState* {
8            if( feof(stream) || ferror(stream) ) {
9               fc.stop();
10              return NULL;
11           }
12           EState *s = BZ2_bzCompressInit(blockSize100k, verbosity, workFactor);
13           in_state.getOneBlock(stream,s);
14           return s;
15        } ) &
16        tbb::make_filter<EState*,EState*>( tbb::filter::
              parallel, []( EState*s ) -> EState* {
17           if( s->nblock )
18              CompressOneBlock(s);
19           return s;
20        } ) &
21        tbb::make_filter<EState*,void>( tbb::filter::
              serial_in_order, [&]( EState* s ) {
22           if( s->nblock )
23              out_state.putOneBlock(s);
24           FreeEState(s);
25        } )
26     );
27     …
28 }
```

LISTING 12.2

Use of TBB `parallel_pipeline` to coordinate bzip2 actions.

signal the end of input. Invocation of method `stop()` on it tells `parallel_pipeline` that no more items will be produced and that the value returned from the functor should be ignored.

12.3 FOUR-STAGE PIPELINE USING TBB

Breaking the last stage into two stages, one serial and one parallel, can enable more parallelism. At the time of this writing, it's not worth the trouble because the output device is typically a disk that works best when accessed serially; otherwise, time is wasted bouncing the disk head around. However, future storage technologies such as solid-state disk might make the following scheme practical.

The scheme is to separate determining where to write an output block from the actual writing. The "where" part is serial, because it relies on knowing how big the previous blocks are. But the actual writing of an output blocks can be done in parallel, and physically in parallel on some output devices. The scheme replaces the last stage of Figure 12.1 with the two stages shown in Figure 12.2. The top stage is serial and does the operations that involve carrying information between blocks. The bottom stage is parallel and does the realignment and writing.

12.4 THREE-STAGE PIPELINE USING CILK PLUS

As explained in Section 9.4.2, Cilk Plus does not support the general pipeline pattern but can synthesize a serial–parallel–serial pipeline via a **consumer reducer**. Listing 12.3 shows the instance of this approach for bzip2 using the same `InputState` and `OutputState` classes as for the TBB implementation. The organization of routines mimics Listing 9.3 on page 259.

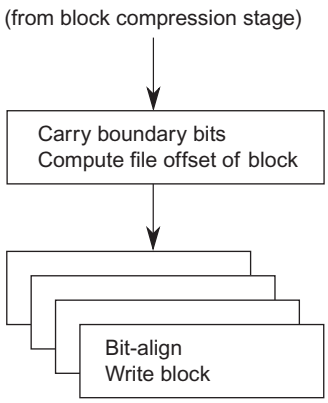

FIGURE 12.2

Possible replacement for last stage of Figure 12.1 that permits parallel writing.

```
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 }
```

LISTING 12.3

Sketch of bzip2 pipeline in Cilk Plus using a consumer reducer. The code uses the `reducer_consume` template from Listing 9.4.

12.5 SUMMARY

The bzip2 program partitions a file into blocks and compresses each one separately. Initial and final processing of each block is sequential. Hence, it is well suited to a serial–parallel–serial pipeline structure. The TBB template `parallel_pipeline` directly supports such a pipeline. Cilk Plus has no direct support for pipelines but the application can be parallelized nonetheless by spawning tasks and assembling the output in order with a consumer reducer.