Exploiting Data Parallelism in Ordered Data Streams

<keywords: threading, data parallelism, I/O, order dependence>

Abstract

Many compute-intensive applications involve complex transformations of ordered input data to ordered output data. Examples include sound and video transcoding, lossless data compression, and seismic data processing. While the algorithms employed in these transformations are often parallel, managing the I/O order dependence can be a challenge. This article identifies some of these challenges and illustrates strategies for addressing them while maintaining parallel performance.

This article is part of the larger series, "The Intel Guide for Developing Multithreaded Applications," which provides guidelines for developing efficient multithreaded applications for Intel® platforms.

Background

Consider the problem of threading a video compression engine designed to perform real-time processing of uncompressed video from a live video source to disk or a network client. Clearly, harnessing the power of multiple processors is a key requirement in meeting the real-time requirements of such an application.

Video compression standards such as MPEG2 and MPEG4 are designed for streaming over unreliable links. Consequently, it easy to treat a single video stream as a sequence of smaller, standalone streams. One can achieve substantial speedups by processing these smaller streams in parallel. Some of the challenges in exploiting this parallelism through multithreading include the following:

- Defining non-overlapping subsets of the problem and assigning them to threads
- Ensuring that the input data is read exactly once and in the correct order
- Outputting blocks in the correct order, regardless of the order in which processing actually completes and without significant performance penalties
- Performing the above without a priori knowledge of the actual extent of the input data

In other situations, such as lossless data compression, it is often possible to determine the input data size in advance and explicitly partition the data into independent input blocks. The techniques outlined here apply equally well to this case.

Advice

The temptation might be to set up a chain of producers and consumers, but this approach is not scalable and is vulnerable to load imbalance. Instead, this article addresses each of the challenges above to achieve a more scalable design using data decomposition.

The approach taken here is to create a team of threads, with each thread reading a block of video, encoding it, and outputting it to a reorder buffer. Upon completion of each block, a thread returns to read and process the next block of video, and so on. This dynamic allocation of work minimizes load imbalance. The reorder buffer ensures that blocks of coded video are written in the correct order, regardless of their order of completion.
The original video encoding algorithm might take this form:

```plaintext
inFile = OpenFile ()
outFile == InitializeOutputFile ()
WriteHeader (outFile)
outputBuffer = AllocateBuffer (bufferSize)
while (frame = ReadNextFrame (inFile))
{
    EncodeFrame (frame, outputBuffer)
    if (outputBuffer size > bufferThreshold)
        FlushBuffer(outputBuffer, outFile)
}
FlushBuffer (outputBuffer, outFile)
```

The first task is to replace the read and encode frame sequence with a block-based algorithm, setting up the problem for decomposition across a team of threads:

```plaintext
WriteHeader (outFile)
while (block = ReadNextBlock (inFile))
{
    while (frame = ReadNextFrame (block))
    {
        EncodeFrame (frame, outputBuffer)
        if (outputBuffer size > bufferThreshold)
            FlushBuffer(outputBuffer, outFile)
    }
    FlushBuffer (outputBuffer, outFile)
}
```

The definition of a block of data will vary from one application to another, but in the case of a video stream, a natural block boundary might be the first frame at which a scene change is detected in the input, subject to constraints of minimum and maximum block sizes. Block-based processing requires allocation of an input buffer and minor changes to the source code to fill the buffer before processing. Likewise, the `readNextFrame` method must be changed to read from the buffer rather than the file.

The next step is to change the output buffering strategy to ensure that entire blocks are written as a unit. This approach simplifies output reordering substantially, since it is necessary only to ensure that the blocks are output in the correct order. The following code reflects the change to block-based output:

```plaintext
WriteHeader (outFile)
while (block = ReadNextBlock (inFile))
{
    while (frame = ReadNextFrame (block))
    {
        EncodeFrame (frame, outputBuffer)
    }
    FlushBuffer (outputBuffer, outFile)
}
```

Depending on the maximum block size, a larger output buffer may be required.

Because each block is independent of the others, a special header typically begins each output block. In the case of an MPEG video stream, this header precedes a complete frame, known as
an I-frame, relative to which future frames are defined. Consequently, the header is moved inside the loop over blocks:

```c
while (block = ReadNextBlock (inFile))
{
  WriteHeader (outputBuffer)
  while (frame = ReadNextFrame (block))
  {
    EncodeFrame (frame, outputBuffer)
  }
  FlushBuffer (outputBuffer, outFile)
}
```

With these changes, it is possible to introduce parallelism using a thread library (i.e., Pthreads or the Win32 threading API) or OpenMP.

```
// Create a team of threads with private
// copies of outputBuffer, block, and frame
// and shared copies of inFile and outFile
while (AcquireLock,
  block = ReadNextBlock (inFile),
  ReleaseLock, block)
{
  WriteHeader (outputBuffer)
  while (frame = ReadNextFrame (block))
  {
    EncodeFrame (frame, outputBuffer)
  }
  FlushBuffer (outputBuffer, outFile)
}
```

This is a simple but effective strategy for reading data safely and in order. Each thread acquires a lock, reads a block of data, then releases the lock. Sharing the input file ensures that blocks of data are read in order and exactly once. Because a ready thread always acquires the lock, the blocks are allocated to threads on a dynamic, or first-come-first-served basis, which typically minimizes load imbalance.

The final task is to ensure that blocks are output safely and in the correct order. A simple strategy would be to use locks and a shared output file to ensure that only one block is written at a time. This approach ensures thread-safety, but would allow the blocks to be output in something other than the original order. Alternately, threads could wait until all previous blocks have been written before flushing their output. Unfortunately, this approach introduces inefficiency because a thread sits idle waiting for its turn to write.

A better approach is to establish a circular reorder buffer for output blocks. Each block is assigned a sequential serial number. The “tail” of the buffer establishes the next block to be written. If a thread finishes processing a block of data other than that pointed to by the tail, it simply enqueues its block in the appropriate buffer position and returns to read and process the next available block. Likewise, if a thread finds that its just-completed block is that pointed to by the tail, it writes that block and any contiguous blocks that were previously enqueued. Finally, it updates the buffer’s tail to point to the next block to be output. The reorder buffer allows completed blocks to be enqueued out-of-order, while ensuring they are written in order.
Figure 1 illustrates one possible state of the reorder buffer. Blocks 0 through 35 have already been processed and written, while blocks 37, 38, 39, 40 and 42 have been processed and are enqueued for writing. When the thread processing block 36 completes, it writes out blocks 36 through 40, leaving the reorder buffer in the state shown in Figure 2. Block 42 remains enqueued until block 41 completes.

Figure 2. State of example reorder buffer after writing.

Naturally, one needs to take certain precautions to ensure the algorithm is robust and fast:

- The shared data structures must be locked when read or written.
- The number of slots in the buffer must exceed the number of threads.
- Threads must efficiently wait, if an appropriate slot is not available in the buffer.
- Pre-allocate multiple output buffers per thread. This allows one to enqueue a pointer to the buffer and avoids extraneous data copies and memory allocations.

Using the output queue, the final algorithm is as follows:

```c
inFile = OpenFile ()
outFile == InitializeOutputFile ()
// Create a team of threads with private
// copies of outputBuffer, block, and frame, shared
// copies of inFile and outFile.
while (AcquireLock,
        block = ReadNextBlock (inFile),
        ReleaseLock, block)
{
    WriteHeader (outputBuffer)
    while (frame = ReadNextFrame (block))
    {
```
EncodeFrame (frame, outputBuffer)
}
QueueOrFlush (outputBuffer, outFile)
}

This algorithm allows in-order I/O but still affords the flexibility of high performance, out-of-order parallel processing.

**Usage Guidelines**

In some instances, the time to read and write data is comparable to the time required to process the data. In these cases, the following techniques may be beneficial:

- Linux* and Windows* provide APIs to initiate a read or write and later wait on or be notified of its completion. Using these interfaces to “pre-fetch” input data and “post-write” output data while performing other computation can effectively hide I/O latency. On Windows, files are opened for asynchronous I/O by providing the FILE_FLAG_OVERLAPPED attribute. On Linux, asynchronous operations are effected through a number of aio_* functions provided by libaio.

- When the amount of input data is significant, static decomposition techniques can lead to physical disk “thrashing”, as the hardware attempts to service a number of concurrent but non-contiguous reads. Following the advice above of a shared file descriptor and a dynamic, first-come-first-served scheduling algorithm can enforce in-order, contiguous reads, which in turn improve overall I/O subsystem throughput.

It is important to carefully choose the size and number of data blocks. Normally, a large number of blocks affords the most scheduling flexibility, which can reduce load imbalance. On the other hand, very small blocks can introduce unnecessary locking overhead and even hinder the effectiveness of data-compression algorithms.

**Additional Resources**

[Intel® Software Network Parallel Programming Community](#)

[OpenMP* Specifications](#)