Expose Parallelism by Avoiding or Removing Artificial Dependencies

<keywords: data dependencies, compiler optimizations, blocking algorithms, Win32 threads, OpenMP, Pthreads>

Abstract

Many applications and algorithms contain serial optimizations that inadvertently introduce data dependencies and inhibit parallelism. One can often remove such dependences through simple transforms, or even avoid them altogether through techniques such as domain decomposition or blocking.

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

While multithreading for parallelism is an important source of performance, it is equally important to ensure that each thread runs efficiently. While optimizing compilers do the bulk of this work, it is not uncommon for programmers to make source code changes that improve performance by exploiting data reuse and selecting instructions that favor machine strengths. Unfortunately, the same techniques that improve serial performance can inadvertently introduce data dependencies that make it difficult to achieve additional performance through multithreading.

One example is the re-use of intermediate results to avoid duplicate computations. As an example, softening an image through blurring can be achieved by replacing each image pixel by a weighted average of the pixels in its neighborhood, itself included. The following pseudo-code describes a 3x3 blurring stencil:

```
for each pixel in (imageIn)
  sum = value of pixel
  // compute the average of 9 pixels from imageIn
  for each neighbor of (pixel)
    sum += value of neighbor
  // store the resulting value in imageOut
  pixelOut = sum / 9
```

The fact that each pixel value feeds into multiple calculations allows one to exploit data reuse for performance. In the following pseudo-code, intermediate results are computed and used three times, resulting in better serial performance:

```
subroutine BlurLine (lineIn, lineOut)
  for each pixel j in (lineIn)
    // compute the average of 3 pixels from line
    // and store the resulting value in lineout
    pixelOut = (pixel j-1 + pixel j + pixel j+1) / 3

declare lineCache[3]
lineCache[0] = 0
BlurLine (line 1 of imageIn, lineCache[1])
for each line i in (imageIn)
  BlurLine (line i+1 of imageIn, lineCache[i mod 3])
```

```
  lineSums = lineCache[0] + lineCache[1] + lineCache[2]
  lineOut = lineSums / 3
```

This optimization introduces a dependence between the computations of neighboring lines of the output image. If one attempts to compute the iterations of this loop in parallel, the dependencies will cause incorrect results.

Another common example is pointer offsets inside a loop:

```
ptr = &someArray[0]
for (i = 0; i < N; i++)
{
  Compute (ptr);
  ptr++;
}
```

By incrementing `ptr`, the code potentially exploits the fast operation of a register increment and avoids the arithmetic of computing `someArray[i]` for each iteration. While each call to compute may be independent of the others, the pointer becomes an explicit dependence; its value in each iteration depends on that in the previous iteration.

Finally, there are often situations where the algorithms invite parallelism but the data structures have been designed to a different purpose that unintentionally hinder parallelism. Sparse matrix algorithms are one such example. Because most matrix elements are zero, the usual matrix representation is often replaced with a "packed" form, consisting of element values and relative offsets, used to skip zero-valued entries.

This article presents strategies to effectively introduce parallelism in these challenging situations.

Advice

Naturally, it's best to find ways to exploit parallelism without having to remove existing optimizations or make extensive source code changes. Before removing any serial optimization to expose parallelism, consider whether the optimization can be preserved by applying the existing kernel to a subset of the overall problem. Normally, if the original algorithm contains parallelism, it is also possible to define subsets as independent units and compute them in parallel.

To efficiently thread the blurring operation, consider subdividing the image into sub-images, or blocks, of fixed size. The blurring algorithm allows the blocks of data to be computed independently. The following pseudo-code illustrates the use of image blocking:

```
// One time operation:
// Decompose the image into non-overlapping blocks.
blockList = Decompose (image, xRes, yRes)

foreach (block in blockList)
{
  BlurBlock (block, imageIn, imageOut)
}
```

The existing code to blur the entire image can be reused in the implementation of `BlurBlock`. Using OpenMP or explicit threads to operate on multiple blocks in parallel yields the benefits of multithreading and retains the original optimized kernel.

In other cases, the benefit of the existing serial optimization is small compared to the overall cost of each iteration, making blocking unnecessary. This is often the case when the iterations are sufficiently coarse-grained to expect a speedup from parallelization. The pointer increment example is one such instance. The induction variables can be easily replaced with explicit indexing, removing the dependence and allowing simple parallelization of the loop.

```
ptr = &someArray[0]
for (i = 0; i < N; i++)
{
  Compute (ptr[i]);
}
```

Note that the original optimization, though small, is not necessarily lost. Compilers often optimize index calculations aggressively by utilizing increment or other fast operations, enabling the benefits of both serial and parallel performance.

Other situations, such as code involving packed sparse matrices, can be more challenging to thread. Normally, it is not practical to unpack data structures but it is often possible to subdivide the matrices into blocks, storing pointers to the beginning of each block. When these matrices are paired with appropriate block-based algorithms, the benefits of a packed representation and parallelism can be simultaneously realized.

The blocking techniques described above are a case of a more general technique called "domain decomposition." After decomposition, each thread works independently on one or more domains. In some situations, the nature of the algorithms and data dictate that the work per domain will be nearly constant. In other situations, the amount of work may vary from domain to domain. In these cases, if the number of domains equals the number of threads, parallel performance can be limited by load imbalance. In general, it is best to ensure that the number of domains is reasonably large compared to the number of threads. This will allow techniques such as dynamic scheduling to balance the load across threads.

Usage Guidelines

Some serial optimizations deliver large performance gains. Consider the number of processors being targeted to ensure that speedups from parallelism will outweigh the performance loss associated with optimizations that are removed.

Introducing blocking algorithms can sometimes hinder the compiler's ability to distinguish aliased from unaliased data. If, after blocking, the compiler can no longer determine that data is unaliased, performance may suffer. Consider using the `restrict` keyword to explicitly prohibit aliasing. Enabling inter-procedural optimizations also helps the compiler detect unaliased data.

Additional Resources

Intel® Software Network Parallel Programming Community

OpenMP* Specifications