Loop Modifications to Enhance Data-Parallel Performance

<keywords: threading, performance, optimization, OpenMP, Threading Building Blocks>

Abstract

In data-parallel applications, the same independent operation is performed repeatedly on different data. Loops are usually the most compute-intensive segments of data parallel applications, so loop optimizations directly impact performance. When confronted with nested loops, the granularity of the computations that are assigned to threads will directly affect performance. Loop transformations such as splitting (loop fission) and merging (loop fusion) nested loops can make parallelization easier and more productive.

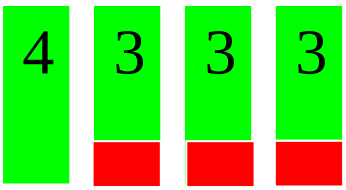
This article is part of the larger series, "The Intel Guide for Developing Multithreaded Applications," which provides guidelines for developing efficient multithreaded applications for Intel® platforms.

Background

Loop optimizations offer a good opportunity to improve the performance of data-parallel applications. These optimizations, such as loop fusion, loop interchange, and loop unrolling, are usually targeted at improving granularity, load balance, and data locality, while minimizing synchronization and other parallel overheads. As a rule of thumb, loops with high iteration counts are typically the best candidates for parallelization, especially when using a relatively small number of threads. A higher iteration count enables better load balance due to the availability of a larger number of tasks that can be distributed among the threads. Still, the amount of work to be done per iteration should also be considered. Unless stated otherwise, the discussion in this section assumes that the amount of computation within each iteration of a loop is (roughly) equal to every other iteration in the same loop.

Consider the scenario of a loop using the OpenMP* for worksharing construct shown in the example code below. In this case, the low iteration count leads to a load imbalance when the loop iterations are distributed over four threads. If a single iteration takes only a few microseconds, this imbalance may not cause a significant impact. However, if each iteration takes an hour, three of the threads remain idle for 60 minutes while the fourth completes. Contrast this to the same loop with 1003 one-hour iterations and four threads. In this case, a single hour of idle time after ten days of execution is insignificant.

```
#pragma omp for
for (i = 0; i < 13; i++)
{…}
```



Advice

For multiple nested loops, choose the outermost loop that is safe to parallelize. This approach generally gives the coarsest granularity. Ensure that work can be evenly distributed to each thread. If this is not possible because the outermost loop has a low iteration count, an inner loop with a large iteration count may be a better candidate for threading. For example, consider the following code with four nested loops:

```
void processQuadArray (int imx, int jmx, int kmx,
  double**** w, double**** ws)
{
  for (int nv = 0; nv < 5; nv++)
    for (int k = 0; k < kmx; k++)
      for (int j = 0; j < jmx; j++)
        for (int i = 0; i < imx; i++)
          ws[nv][k][j][i] = Process(w[nv][k][j][i]);
}
```

With any number other than five threads, parallelizing the outer loop will result in load imbalance and idle threads. The inefficiency would be especially severe if the array dimensions `imx`, `jmx`, and `kmx` are very large. Parallelizing one of the inner loops is a better option in this case.

Avoid the implicit barrier at the end of worksharing constructs when it is safe to do so. All OpenMP worksharing constructs (`for`, `sections`, `single`) have an implicit barrier at the end of the structured block. All threads must rendezvous at this barrier before execution can proceed. Sometimes these barriers are unnecessary and negatively impact performance. Use the OpenMP `nowait` clause to disable this barrier, as in the following example:

```
void processQuadArray (int imx, int jmx, int kmx,
  double**** w, double**** ws)
{
  #pragma omp parallel shared(w, ws)
  {
    int nv, k, j, i;
    for (nv = 0; nv < 5; nv++)
      for (k = 0; k < kmx; k++) // kmx is usually small
        #pragma omp for shared(nv, k) nowait
          for (j = 0; j < jmx; j++)
            for (i = 0; i < imx; i++)
              ws[nv][k][j][i] = Process(w[nv][k][j][i]);
  }
}
```

Since the computations in the innermost loop are all independent, there is no reason for threads to wait at the implicit barrier before going on to the next `k` iteration. If the amount of work per iteration is unequal, the `nowait` clause allows threads to proceed with useful work rather than sit idle at the implicit barrier.

If a loop has a loop-carried dependence that prevents the loop from being executed in parallel, it may be possible to break up the body of the loop into separate loops than can be executed in parallel. Such division of a loop body into two or more loops is known as “loop fission”. In the following example, loop fission is performed on a loop with a dependence to create new loops that can execute in parallel:

```
float *a, *b;
int i;
for (i = 1; i < N; i++) {
  if (b[i] > 0.0)
    a[i] = 2.0 * b[i];
  else
    a[i] = 2.0 * fabs(b[i]);
  b[i] = a[i-1];
}
```

The assignment of elements within the `a` array are all independent, regardless of the sign of the corresponding elements of `b`. Each assignment of an element in b is independent of any other assignment, but depends on the completion of the assignment of the required element of `a`. Thus, as written, the loop above cannot be parallelized.

By splitting the loop into the two independent operations, both of those operations can be executed in parallel. For example, the Intel® Threading Building Blocks (Intel® TBB) `parallel_for` algorithm can be used on each of the resulting loops as seen here:

```
float *a, *b;

parallel_for (1, N, 1,
  [&](int i) {
    if (b[i] > 0.0)
      a[i] = 2.0 * b[i];
    else
      a[i] = 2.0 * fabs(b[i]);
    });
parallel_for (1, N, 1,
  [&](int i) {
    b[i] = a[i-1];
  });
```

The return of the first `parallel_for` call before execution of the second ensures that all the updates to the `a` array have completed before the updates on the b array are started.

Another use of loop fission is to increase data locality. Consider the following sieve-like code :

```
for (i = 0; i < list_len; i++)
  for (j = prime[i]; j < N; j += prime[i])
    marked[j] = 1;
```

The outer loop selects the starting index and step of the inner loop from the prime array. The inner loop then runs through the length of the marked array depositing a '1' value into the chosen elements. If the marked array is large enough, the execution of the inner loop can evict cache lines from the early elements of `marked` that will be needed on the subsequent iteration of the outer loop. This behavior will lead to a poor cache hit rate in both serial and parallel versions of the loop.

Through loop fission, the iterations of the inner loop can be broken into chunks that will better fit into cache and reuse the cache lines once they have been brought in. To accomplish the fission in this case, another loop is added to control the range executed over by the innermost loop:

```
for (k = 0; k < N; k += CHUNK_SIZE)
  for (i = 0; i < list_len; i++) {
    start = f(prime[i], k);
    end = g(prime[i], k);
    for (j = start; j < end; j += prime[i])
      marked[j] = 1;
  }
```

For each iteration of the outermost loop in the above code, the full set of iterations of the i-loop will execute. From the selected element of the prime array, the start and end indices within the

chunk of the `marked` array (controlled by the outer loop) must be found. These computations have been encapsulated within the `f()` and `g()` routines. Thus, the same chunk of `marked` will be processed before the next one is processed. And, since the processing of each chunk is independent of any other, the iteration of the outer loop can be made to run in parallel.

Merging nested loops to increase the iteration count is another optimization that may aid effective parallelization of loop iterations. For example, consider the code on the left with two nested loops having iteration counts of 23 and 1000, respectively. Since 23 is prime, there is no way to evenly divide the outer loop iterations; also, 1000 iteration may not be enough work to sufficiently minimize the overhead of threading only the inner loop. On the other hand, the loops can be fused into a single loop with 23,000 iterations (as seen on the right), which could alleviate the problems with parallelizing the original code.

```
#define N 23
#define M 1000
. . .
for (k = 0; k < N; k++)
  for (j = 0; j < M; j++)
    wn[k][j] = Work(w[k][j], k,
j);
```

```
#define N 23
#define M 1000
. . .
for (kj = 0; kj < N*M; kj++) {
  k = kj / M;
  j = kj % M;
  wn [k][j] = Work(w[k][j], k,
j);
}
```

However, if the iteration variables are each used within the loop body (e.g., to index arrays), the new loop counter must be translated back into the corresponding component values, which creates additional overhead that the original algorithm did not have.

Fuse (or merge) loops with similar indices to improve granularity and data locality and to minimize overhead when parallelizing. The first two loops in the left-hand example code can be easily merged:

```
for (j = 0; j < N; j++)
  a[j] = b[j] + c[j];

for (j = 0; j < N; j++)
  d[j] = e[j] + f[j];

for (j = 5; j < N - 5; j++)
  g[j] = d[j+1] + a[j+1];
```

```
for (j = 0; j < N; j++)
{
  a[j] = b[j] + c[j];
  d[j] = e[j] + f[j];
}

for (j = 5; j < N - 5; j++)
  g[j] = d[j+1] + a[j+1];
```

Merging these loops increases the amount of work per iteration (i.e., granularity) and reduces loop overhead. The third loop is not easily merged because its iteration count is different. More important, however, a data dependence exists between the third loop and the previous two loops.

Use the OpenMP `if` clause to choose serial or parallel execution based on runtime information. Sometimes the number of iterations in a loop cannot be determined until runtime. If there is a negative performance impact for executing an OpenMP parallel region with multiple threads (e.g., a small number of iterations), specifying a minimum threshold will help maintain performance, as in the following example:

```
#pragma omp parallel for if(N >= threshold)
  for (i = 0; i < N; i++) { … }
```

For this example code, the loop is only executed in parallel if the number of iterations exceeds the threshold specified by the programmer.

Since there is no equivalent in Intel TBB, an explicit conditional test could be done to determine if a parallel or serial execution of code should be done. Alternately, a parallel algorithm could be called and the Intel TBB task scheduler could be given free rein to determine that a single thread should be used for low enough values of `N`. There would be some overhead required for this last option.

Additional Resources

Intel® Software Network Parallel Programming Community

OpenMP* Specifications

Intel® Threading Building Blocks

Intel Threading Building Blocks for Open Source

James Reinders, *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly Media, Inc. Sebastopol, CA, 2007.