



Intel® Guide for Developing Multithreaded Applications

Part 1: Application Threading and Synchronization

Summary

The Intel® Guide for Developing Multithreaded Applications provides a reference to design and optimization guidelines for developing efficient multithreaded applications across Intel-based symmetric multiprocessors (SMP) and/or systems with Intel® Hyper-Threading Technology. An application developer can source this technical guide to improve multithreading performance and minimize unexpected performance variations on current and future SMP architectures built with Intel processors.

Part 1 is a collection of technical papers providing software developers with the most current technical information on application threading and synchronization techniques. Part 2 covers different memory management approaches and programming tools that help speed development and streamline parallel programming.

Each topic contains a standalone discussion of important threading issues, and complementary topics are cross-referenced throughout the guide for easy links to deeper knowledge.

This guide provides general advice on multithreaded performance; is not intended to serve as a textbook on multithreading nor is it a porting guide to Intel platforms. Hardware-specific optimizations were deliberately kept to a minimum. Future versions of this guide will include topics covering hardware-specific optimizations for developers willing to sacrifice portability for higher performance.

For online versions of individual articles, see:
[Intel Guide for Developing Multithreaded Applications](#)

Get more news for developers at:
[Intel® Software Network News](#)

Brought to you by
[Intel® Software Dispatch](#)
Delivering the latest tools, techniques, and best practices for leading-edge software infrastructure, platform software, and developer resources.

Prerequisites

Readers should have programming experience in a high-level language, preferably C, C++, and/or Fortran, though many recommendations in this document also apply to Java, C#, and Perl. Readers must also understand basic concurrent programming and be familiar with one or more threading methods, preferably OpenMP*, POSIX threads (also referred to as Pthreads), or the Win32* threading API.

Developers are invited participate in online discussions on these topics at:
[Threading on Intel® Parallel Architectures forum.](#)

Table of Contents

Application Threading

This chapter covers general topics in parallel performance but occasionally refers to API-specific issues.

1.1 - Predicting and Measuring Parallel Performance.....	p. 3
1.2 - Loop Modifications to Enhance Data-Parallel Performance	p. 6
1.3 - Granularity and Parallel Performance	p. 11
1.4 - Load Balance and Parallel Performance	p. 16
1.5 - Expose Parallelism by Avoiding or Removing Artificial Dependencies.....	p. 19
1.6 - Using Tasks Instead of Threads	p. 22
1.7 - Exploiting Data Parallelism in Ordered Data Streams	p. 25

Synchronization

The topics in this chapter discuss techniques to mitigate the negative impact of synchronization on performance.

2.1 - Managing Lock Contention- Large and Small Critical Sections.....	p. 30
2.2 - Use Synchronization Routines Provided by the Threading API Rather than Hand-Coded Synchronization	p. 34
2.3 - Choosing Appropriate Synchronization Primitives to Minimize Overhead	p. 37
2.4 - Use Non-blocking Locks When Possible.....	p. 41

1.1 – Predicting and Measuring Parallel Performance

Abstract

Building parallel versions of software can enable applications to run a given data set in less time, run multiple data sets in a fixed amount of time, or run large-scale data sets that are prohibitive with unthreaded software. The success of parallelization is typically quantified by measuring the speedup of the parallel version relative to the serial version. In addition to that comparison, however, it is also useful to compare that speedup relative to the upper limit of the potential speedup. That issue can be addressed using Amdahl's Law and Gustafson's Law.

Background

The faster an application runs, the less time a user will need to wait for results. Shorter execution time also enables users to run larger data sets (e.g., a larger number of data records, more pixels, or a bigger physical model) in an acceptable amount of time. One computed number that offers a tangible comparison of serial and parallel execution time is *speedup*.

Simply stated, speedup is the ratio of serial execution time to parallel execution time. For example, if the serial application executes in 6720 seconds and a corresponding parallel application runs in 126.7 seconds (using 64 threads and cores), the speedup of the parallel application is 53X ($6720/126.7 = 53.038$).

For an application that scales well, the speedup should increase at or close to the same rate as the increase in the number of cores (threads). When increasing the number of threads used, if measured speedups fail to keep up, level out, or begin to go down, the application doesn't scale well on the data sets being measured. If the data sets are typical of actual data sets on which the application will be executed, the application won't scale well.

Related to speedup is the metric of *efficiency*. While speedup is a metric to determine how much faster parallel execution is versus serial execution, efficiency indicates how well software utilizes the computational resources of the system. To calculate the efficiency of parallel execution, take the observed speedup and divide by the number of cores used. This number is then expressed as a percentage. For example, a 53X speedup on 64 cores equates to an efficiency of 82.8% ($53/64 = 0.828$). This means that, on average, over the course of the execution, each of the cores is idle about 17% of the time.

Amdahl's Law

Before starting a parallelization project, developers may wish to estimate the amount of performance increase (speedup) that they can realize. If the percentage of serial code execution that could be executed in parallel is known (or estimated), one can use Amdahl's Law to compute an upper bound on the speedup of an application without actually writing any concurrent code. Several variations of the Amdahl's Law formula have been put forth in the literature. Each uses the percentage of (proposed) parallel execution time (pctPar), serial execution time (1 - pctPar), and the number of threads/cores (p). A simple formulation of Amdahl's Law to estimate speedup of a parallel application on p cores is given here:

$$Speedup \leq \frac{1}{(1 - pctPar) + \frac{pctPar}{p}}$$

The formula is simply the serial time, normalized to 1, divided by the estimated parallel execution time, using percentages of the

normalized serial time. The parallel execution time is estimated to be the percentage of serial execution ($1 - \text{pctPar}$) and the percentage of execution that can be run in parallel divided by the number of cores to be used (pctPar/p). For example, if 95% of a serial application's run time could be executed in parallel on eight cores, the estimated speedup, according to Amdahl's Law, could as much 6X ($1 / (0.05 + 0.95/8) = 5.925$).

In addition to the less than or equal relation (\geq) in the formula, the formulations of Amdahl's Law assume that those computations that can be executed in parallel will be divisible by an infinite number of cores. Such an assumption effectively removes the second term in the denominator, which means that the most speedup possible is simply the inverse of the percentage of remaining serial execution.

Amdahl's Law has been criticized for ignoring real-world overheads such as communication, synchronization, and other thread management, as well as the assumption of infinite-core processors. In addition to not taking into account the overheads inherent in concurrent algorithms, one of the strongest criticisms of Amdahl's Law is that as the number of cores increases, the amount of data handled is likely to increase as well. Amdahl's Law assumes a fixed data set size for whatever number of cores is used and that the percentage of overall serial execution time will remain the same.

Gustafson's Law

If a parallel application using eight cores were able to compute a data set that was eight times the size of the original, does the execution time of the serial portion increase? Even if it does, it does not grow in the same proportion as the data set. Real-world data suggests that the serial execution time will remain almost constant.

Gustafson's Law, also known as scaled speedup, takes into account an increase in the data size in proportion to the increase in the number of cores and computes the (upper bound) speedup of the application, as if the larger data set could be executed in serial. The formula for scaled speedup is as follows:

$$\text{Speedup} \leq p + (1 - p)s.$$

As with the formula for Amdahl's Law, p is the number of cores. To simplify the notation, s is the percentage of serial execution time in the parallel application for a given data set size. For example, if 1% of execution time on 32 cores will be spent in serial execution, the speedup of this application over the same data set being run on a single core with a single thread (assuming that to be possible) is:

$$\text{Speedup} \leq 32 + (1 - 32)(0.01) = 32 - 0.31 = 31.69X.$$

Consider what Amdahl's Law would estimate for the speedup with these assumptions. Assuming the serial execution percentage to be 1%, the equation for Amdahl's Law yields $1/(0.01 + (0.99/32)) = 24.43X$. This is a false computation, however, since the given percentage of serial time is relative to the 32-core execution. The details of this example do not indicate what the corresponding serial execution percentage would be for more cores or fewer cores (or even one core). If the code is perfectly scalable and the data size is scaled with the number of cores, then this percentage could remain constant, and the Amdahl's Law computation would be the predicted speedup of the (fixed-size) single-core problem on 32 cores.

On the other hand, if the total parallel application execution time is known in the 32-core case, the fully serial execution time can be calculated and the speed up for that fix-sized problem (further assuming that it could be computed with a single core) could be predicted with Amdahl's Law on 32 cores. Assuming the total execution time for a parallel application is 1040 seconds on 32 cores, then 1% of that time would be serial only, or 10.4 seconds. By multiplying the number of seconds (1029.6) for parallel execution on 32 cores, the total amount of work done by the application takes $1029.6 \times 32 + 10.4 = 32957.6$ seconds. The nonparallel time (10.4 seconds) is 0.032% of that total work time. Using that figure, Amdahl's Law calculates a speedup of

$$1/(0.00032 + (0.99968/32)) = 31.686X.$$

Since the percentage of serial time within the parallel execution must be known to use Gustafson's Law, a typical usage for this formula is to compute the speedup of the scaled parallel execution (larger data sets as the number of cores increases) to the serial execution of the same sized problem. From the above examples, a strict use of the data about the application executions within the formula for Amdahl's Law gives a much more pessimistic estimate than the scaled speedup formula.

Advice

When computing speedup, the best serial algorithm and fastest serial code must be compared. Frequently, a less than optimal serial algorithm will be easier to parallelize. Even in such a case, it is unlikely that anyone would use serial code when a faster serial version exists. Thus, even though the underlying algorithms are different, the best serial run time from the fastest serial code must be used to compute the speedup for a comparable parallel application.

When stating a speedup value, a multiplier value should be used. In the past, the speedup ratio has been expressed as a percentage. In this context, using percentages can lead to confusion. For example, if it were stated that a parallel code is 200% faster than the serial code, does it run in half the time of the serial version or one-third of the time? Is 105% speedup almost the same time as the serial execution or more than twice as fast? Is the baseline serial time 0% speedup or 100% speedup? On the other hand, if the parallel application were reported to have a speedup of 2X, it is clear that it took half the time (i.e., the parallel version could have executed twice in the same time it took the serial code to execute once).

In very rare circumstances, the speedup of an application exceeds the number of cores. This phenomenon is known as super-linear speedup. The typical cause for super-linear speedup is that decomposition of the fixed-size data set has become small enough per core to fit into local cache. When running in serial, the data had to stream through cache, and the processor was made to wait while cache lines were fetched. If the data was large enough to evict some previously used cache lines, any subsequent reuse of those early cache lines would cause the processor to wait once more for cache lines. When the data is divided into chunks that all fit into the cache on a core, there is no waiting for reused cache lines once they have all been placed in the cache. Thus, the use of multiple cores can eliminate some of the overhead associated with the serial code executing on a single core. Data sets that are too small-smaller than a typical data set size-can give a false sense of performance improvement.

Usage Guidelines

Other parallel execution models have been proposed that attempt to make reasonable assumptions for the discrepancies in the simple model of Amdahl's Law.

Still, for its simplicity and the understanding by the user that this is a theoretical upper bound, which is very unlikely to be achieved or surpassed, Amdahl's Law is a simple and valuable indication of the potential for speedup in a serial application.

Additional Resources

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

John L. Gustafson. "Reevaluating Amdahl's Law." *Communications of the ACM*, Vol. 31, pp. 532-533, 1988.

Michael J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill, 2004.

1.2 – Loop Modifications to Enhance Data-Parallel Performance

Abstract

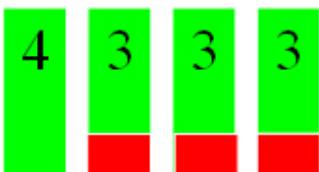
In data-parallel applications, the same independent operation is performed repeatedly on different data. Loops are usually the most compute-intensive segments of data parallel applications, so loop optimizations directly impact performance. When confronted with nested loops, the granularity of the computations 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.

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. 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.

```
- collapse source  view plain  copy to clipboard  print  ?
01.  #pragma omp for
02.  for (i = 0; i < 13; i++)
03.  { ... }
04.
```



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:

```

- collapse source view plain copy to clipboard print ?
01. void processQuadArray (int imx, int jmx, int kmx,
02. double**** w, double**** ws)
03. {
04.     for (int nv = 0; nv < 5; nv++)
05.         for (int k = 0; k < kmx; k++)
06.             for (int j = 0; j < jmx; j++)
07.                 for (int i = 0; i < imx; i++)
08.                     ws[nv][k][j][i] = Process(w[nv][k][j][i]);
09. }
10.

```

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:

```

- collapse source view plain copy to clipboard print ?
01. void processQuadArray (int imx, int jmx, int kmx,
02. double**** w, double**** ws)
03. {
04.     #pragma omp parallel shared(w, ws)
05.     {
06.         int nv, k, j, i;
07.         for (nv = 0; nv < 5; nv++)
08.             for (k = 0; k < kmx; k++) // kmx is usually small
09.                 #pragma omp for shared(nv, k) nowait
10.                 for (j = 0; j < jmx; j++)
11.                     for (i = 0; i < imx; i++)
12.                         ws[nv][k][j][i] = Process(w[nv][k][j][i]);
13.     }
14. }
15.

```

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 that 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:

- collapse source view plain copy to clipboard print ?

```
01. float *a, *b;
02. int i;
03. for (i = 1; i < N; i++) {
04.     if (b[i] > 0.0)
05.         a[i] = 2.0 * b[i];
06.     else
07.         a[i] = 2.0 * fabs(b[i]);
08.     b[i] = a[i-1];
09. }
10.
```

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:

- collapse source view plain copy to clipboard print ?

```
01. float *a, *b;
02.
03. parallel_for (1, N, 1,
04.     [&] (int i) {
05.         if (b[i] > 0.0)
06.             a[i] = 2.0 * b[i];
07.         else
08.             a[i] = 2.0 * fabs(b[i]);
09.     });
10. parallel_for (1, N, 1,
11.     [&] (int i) {
12.         b[i] = a[i-1];
13.     });
14.
```

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:

- collapse source view plain copy to clipboard print ?

```
01. for (i = 0; i < list_len; i++)
02.     for (j = prime[i]; j < N; j += prime[i])
03.         marked[j] = 1;
04.
```

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:


```

- collapse source  view plain  copy to clipboard  print  ?
01.  for (k = 0; k < N; k += CHUNK_SIZE)
02.      for (i = 0; i < list_len; i++) {
03.          start = f(prime[i], k);
04.          end = g(prime[i], k);
05.          for (j = start; j < end; j += prime[i])
06.              marked[j] = 1;
07.      }
08.

```

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.

```

- collapse source  view plain  copy to clipboard  print  ?
01.  #define N 23
02.  #define M 1000
03.  . . .
04.  for (k = 0; k < N; k++)
05.      for (j = 0; j < M; j++)
06.          wn[k][j] = Work(w[k][j], k, j);
07.

```

```

- collapse source  view plain  copy to clipboard  print  ?
01.  #define N 23
02.  #define M 1000
03.  . . .
04.  for (kj = 0; kj < N*M; kj++) {
05.      k = kj / M;
06.      j = kj % M;
07.      wn[k][j] = Work(w[k][j], k, j);
08.  }
09.

```

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:

```

- collapse source  view plain  copy to clipboard  print  ?
01.  for (j = 0; j < N; j++)
02.      a[j] = b[j] + c[j];
03.
04.  for (j = 0; j < N; j++)
05.      d[j] = e[j] + f[j];
06.
07.  for (j = 5; j < N - 5; j++)
08.      g[j] = d[j+1] + a[j+1];
09.

```

```

- collapse source  view plain  copy to clipboard  print  ?
01.  for (j = 0; j < N; j++)
02.  {
03.      a[j] = b[j] + c[j];
04.      d[j] = e[j] + f[j];
05.  }
06.
07.  for (j = 5; j < N - 5; j++)
08.      g[j] = d[j+1] + a[j+1];
09.

```

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:

```

- collapse source  view plain  copy to clipboard  print  ?
01.  #pragma omp parallel for if(N >= threshold)
02.      for (i = 0; i < N; i++) { ... }
03.

```

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.

1.3 - Granularity and Parallel Performance

Abstract

One key to good parallel performance is choosing the right granularity for the application. Granularity is the amount of real work in the parallel task. If granularity is too fine, then performance can suffer from communication overhead. If granularity is too coarse, then performance can suffer from load imbalance. The goal is to determine the right granularity (usually larger is better) for parallel tasks, while avoiding load imbalance and communication overhead to achieve the best performance.

Background

The size of work in a single parallel task (granularity) of a multithreaded application greatly affects its parallel performance. When decomposing an application for multithreading, one approach is to logically *partition* the problem into as many parallel tasks as possible. Within the parallel tasks, next determine the necessary *communication* in terms of shared data and execution order. Since partitioning tasks, assigning the tasks to threads, and communicating (sharing) data between tasks are not free operations, one often needs to *agglomerate*, or combine partitions, to overcome these overheads and achieve the most efficient implementation. The agglomeration step is the process of determining the best granularity for parallel tasks.

The granularity is often related to how balanced the work load is between threads. While easier to balance the workload of a large number of smaller tasks, this may cause too much parallel overhead in the form of communication, synchronization, etc. One can reduce parallel overhead by increasing the granularity (amount of work) within each task by combining smaller tasks into a single task. Tools such as the Intel® Parallel Amplifier can help identify the right granularity for an application.

The following examples demonstrate how to improve the performance of a parallel program by decreasing the communication overhead and finding the right granularity for the threads. The example used throughout this article is a prime-number counting algorithm that uses a simple brute force test of all dividing each potential prime by all possible factors until a divisor is found or the number is shown to be a prime. Because positive odd numbers can be computed by either $(4k+1)$ or $(4k+3)$, for $k \geq 0$, the code will also keep a count of the prime numbers that fall into each form. The examples will count all of the prime numbers between 3 and 1 million.

The first variation of the code shows a parallel version using OpenMP*:

```
- collapse source view plain copy to clipboard print ?
01. #pragma omp parallel
02. { int j, limit, prime;
03.   #pragma for schedule(dynamic, 1)
04.   for(i = 3; i <= 1000000; i += 2) {
05.     limit = (int) sqrt((float)i) + 1;
06.     prime = 1; // assume number is prime
07.     j = 3;
08.     while (prime && (j <= limit)) {
09.       if (i%j == 0) prime = 0;
10.       j += 2;
11.     }
12.
13.     if (prime) {
14.       #pragma omp critical
15.       {
16.         numPrimes++;
17.         if (i%4 == 1) numP41++; // 4k+1 primes
18.         if (i%4 == 3) numP43++; // 4k-1 primes
19.       }
20.     }
21.   }
22. }
```

This code has both high communication overhead (in the form of synchronization), and an individual task size that is too small for the threads. Inside the loop, there is a critical region that is used to provide a safe mechanism for incrementing the counting variables. The critical region adds synchronization and lock overhead to the parallel loop as shown by the Intel Parallel Amplifier display in Figure 1.

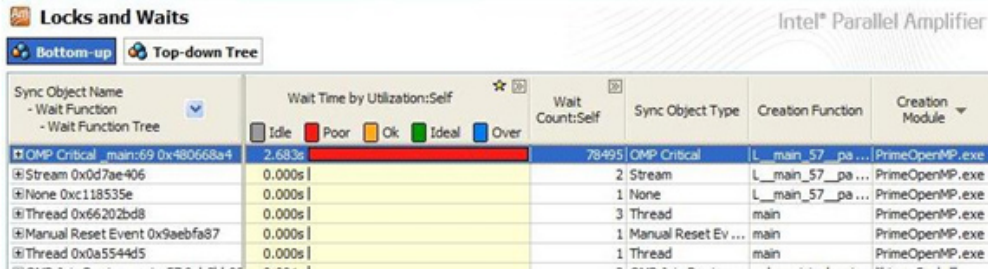


Figure 1. Locks and Waits analysis results demonstrating that the OpenMP* critical region is cause of synchronization overhead.

The incrementing of counter variables based on values within a large dataset is a common expression that is referred to as a reduction. The lock and synchronization overhead can be removed by eliminating the critical region and adding an OpenMP reduction clause:

```

- collapse source  view plain  copy to clipboard  print  ?
01.  #pragma omp parallel
02.  {
03.      int j, limit, prime;
04.
05.      #pragma for schedule(dynamic, 1) \
06.      reduction(+:numPrimes,numP41,numP43)
07.      for(i = 3; i &lt;= 1000000; i += 2) {
08.          limit = (int) sqrt((float)i) + 1;
09.          prime = 1; // assume number is prime
10.          j = 3;
11.          while (prime && (j &lt;= limit))
12.          {
13.              if (i%j == 0) prime = 0;
14.              j += 2;
15.          }
16.
17.          if (prime)
18.          {
19.              numPrimes++;
20.              if (i%4 == 1) numP41++; // 4k+1 primes
21.              if (i%4 == 3) numP43++; // 4k-1 primes
22.          }
23.      }
24.  }

```

Depending on how many iterations are executed for a loop, removal of a critical region within the body of the loop can improve the execution speed by orders of magnitude. However, the code above may still have some parallel overhead. This is caused by the work size for each task being too small. The schedule (dynamic, 1) clause specifies that the scheduler distribute one iteration (or chunk) at a time dynamically to each thread. Each worker thread processes one iteration and then returns to the scheduler, and synchronizes to get another iteration. By increasing the chunk size, we increase the work size for each task that is assigned to a thread and therefore reduce the number of times each thread must synchronize with the scheduler.

While this approach can improve performance, one must bear in mind (as mentioned above) that increasing the granularity too much can cause load imbalance. For example, consider increasing the chunk size to 10000, as in the code below:

```

- collapse source  view plain  copy to clipboard  print  ?
01.  #pragma omp parallel
02.  {
03.      int j, limit, prime;
04.      #pragma for schedule(dynamic, 100000) \
05.          reduction(+:numPrimes, numP41, numP43)
06.      for(i = 3; i <= 1000000; i += 2)
07.      {
08.          limit = (int) sqrt((float)i) + 1;
09.          prime = 1; // assume number is prime
10.          j = 3;
11.          while (prime && (j <= limit))
12.          {
13.              if (i%j == 0) prime = 0;
14.              j += 2;
15.          }
16.
17.          if (prime)
18.          {
19.              numPrimes++;
20.              if (i%4 == 1) numP41++; // 4k+1 primes
21.              if (i%4 == 3) numP43++; // 4k-1 primes
22.          }
23.      }
24.  }

```

Analysis of the execution of this code within Parallel Amplifier shows an imbalance in the amount of computation done by the four threads used, as shown in Figure 2. The key point for this computation example is that each chunk has a different amount of work and there are too few chunks to be assigned as tasks (ten chunks for four threads), which causes the load imbalance. As the value of the potential primes increases (from the for loop), more iterations are required to test all possible factors for prime numbers (in the while loop). Thus, the total work for each chunk will require more iteration of the while loop than the previous chunks.

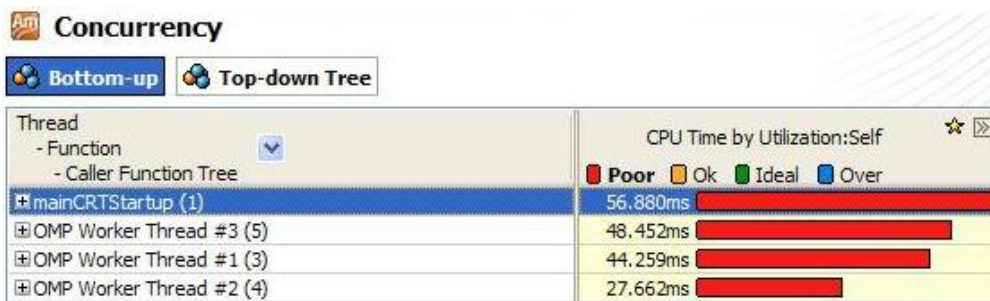


Figure 2. Concurrency analysis results demonstrating the imbalance of execution time used by each thread.

A more appropriate work size (100) should be used to select the right granularity for the program. Also, since the difference in the amount of work between consecutive tasks will be less severe than the previous chunk size, a further elimination of parallel overhead can be accomplished by using the static schedule rather than dynamic. The code below shows the change in the schedule clause that will virtually the overhead from this code segment and produce the fastest overall parallel performance.

```

- collapse source  view plain  copy to clipboard  print  ?
01.  #pragma omp parallel
02.  {
03.      int j, limit, prime;
04.      #pragma for schedule(static, 100) \
05.          reduction(+:numPrimes, numP41, numP43)
06.      for(i = 3; i <= 1000000; i += 2)
07.      {
08.          limit = (int) sqrt((float)i) + 1;
09.          prime = 1; // assume number is prime
10.          j = 3;
11.          while (prime && (j <= limit))
12.          {
13.              if (i%j == 0) prime = 0;
14.              j += 2;
15.          }
16.
17.          if (prime)
18.          {
19.              numPrimes++;
20.              if (i%4 == 1) numP41++; // 4k+1 primes
21.              if (i%4 == 3) numP43++; // 4k-1 primes
22.          }
23.      }
24.  }

```

Advice

Parallel performance of multithreaded code depends on granularity: how work is divided among threads and how communication is accomplished between those threads. Following are some guidelines for improving performance by adjusting granularity:

- Know your application
 - Understand how much work is being done in various parts of the application that will be executed in parallel.
 - Understand the communication requirements of the application. Synchronization is a common form of communication, but also consider the overhead of message passing and data sharing across memory hierarchies (cache, main memory, etc.).
- Know your platform and threading model
 - Know the costs of launching parallel execution and synchronization with the threading model on the target platform.
 - Make sure that the application's work per parallel task is much larger than the overheads of threading.
 - Use the least amount of synchronization possible and use the lowest-cost synchronization possible.
 - Use a partitioner object in Intel® Threading Building Blocks parallel algorithms to allow the task scheduler to choose a good granularity of work per task and load balance on execution threads.
- Know your tools
 - In Intel Parallel Amplifier "Locks and Waits" analysis, look for significant lock, synchronization, and parallel overheads as a sign of too much communication.
 - In Intel Parallel Amplifier "Concurrency" analysis, look for load imbalance as a sign of the granularity being too large or tasks needing a better distribution to threads.

Usage Guidelines

While the examples above make reference to OpenMP frequently, all of the advice and principles described apply to other threading models, such as Windows threads and POSIX* threads. All threading models have overhead associated with their

various functions, such as launching parallel execution, locks, critical regions, message passing, etc. The advice here about reducing communication and increasing work size per thread without increasing load imbalance applies to all threading models. However, the differing costs of the differing models may dictate different choices of granularity.

Additional Resources

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

[Clay Breshears, The Art of Concurrency, O'Reilly Media, Inc., 2009.](#)

[Barbara Chapman, Gabriele Jost, and Ruud van der Post, Using OpenMP: Portable Shared Memory Parallel Programming, The MIT Press, 2007.](#)

[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.](#)

[Ding-Kai Chen, et al, "The Impact of Synchronization and Granularity on Parallel Systems", Proceedings of the 17th Annual International Symposium on Computer Architecture 1990, Seattle, Washington, USA.](#)

1.4 – Load Balance and Parallel Performance

Abstract

Load balancing an application workload among threads is critical to performance. The key objective for load balancing is to minimize idle time on threads. Sharing the workload equally across all threads with minimal work sharing overheads results in fewer cycles wasted with idle threads not advancing the computation, and thereby leads to improved performance. However, achieving perfect load balance is non-trivial, and it depends on the parallelism within the application, workload, the number of threads, load balancing policy, and the threading implementation.

Background

An idle core during computation is a wasted resource, and when effective parallel execution could be running on that core, it increases the overall execution time of a threaded application. This idleness can result from many different causes, such as fetching from memory or I/O. While it may not be possible to completely avoid cores being idle at times, there are measures that programmers can apply to reduce idle time, such as overlapped I/O, memory prefetching, and reordering data access patterns for better cache utilization.

Similarly, idle threads are wasted resources in multithreaded executions. An unequal amount of work being assigned to threads results in a condition known as a “load imbalance.” The greater the imbalance, the more threads will remain idle and the greater the time needed to complete the computation. The more equitable the distribution of computational tasks to available threads, the lower the overall execution time will be.

As an example, consider a set of twelve independent tasks with the following set of execution times: {10, 6, 4, 4, 2, 2, 2, 2, 1, 1, 1, 1}. Assuming that four threads are available for computing this set of tasks, a simple method of task assignment would be to schedule each thread with three total tasks distributed in order. Thus, Thread 0 would be assigned work totaling 20 time units (10+6+4), Thread 1 would require 8 time units (4+2+2), Thread 2 would require 5 time units (2+2+1), and Thread 3 would be able to execute the three tasks assigned in only 3 time units (1+1+1). Figure 1(a) illustrates this distribution of work and shows that the overall execution time for these twelve tasks would be 20 time units (time runs from top to bottom).

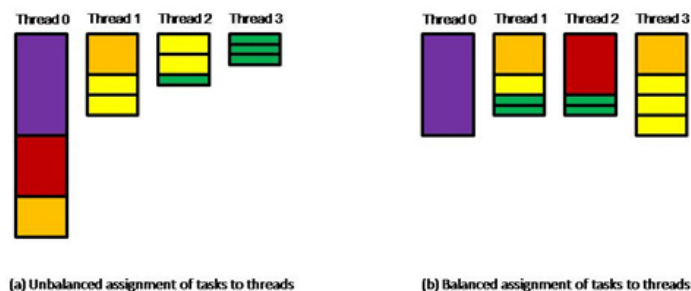


Figure 1. Examples of task distribution among four threads.

A better distribution of work would have been Thread 0: {10}, Thread 1: {4, 2, 1, 1}, Thread 2: {6, 1, 1}, and Thread 3: {4, 2, 2}, as shown in Figure 1(b). This schedule would take only 10 time units to complete and with only have two of the four threads idle for 2 time units each.

Advice

For the case when all tasks are the same length, a simple static division of tasks among available threads-dividing the total number of tasks into (nearly) equal-sized groups assigned to each thread-is an easy and equitable solution. In the general case, however, even when all task lengths are known in advance, finding an optimal, balanced assignment of tasks to threads is an intractable problem. When the lengths of individual tasks are not the same, a better solution may be a more dynamic division of tasks to the assigned threads.

The OpenMP* iterative worksharing construct typically defaults to static scheduling of iterations onto threads (if not, this can scheduling can be specified). When the workload varies among the iterations and the pattern is unpredictable, a dynamic scheduling of iterations to threads can better balance the load. Two scheduling alternatives, dynamic and guided, are specified through the schedule clause. Under dynamic scheduling, chunks of iterations are assigned to threads; when the assignment has been completed, threads request a new chunk of iterations. The optional chunk argument of the schedule clause denotes the fixed size of iteration chunks for dynamic scheduling.

```
- collapse source view plain copy to clipboard print ?
01. #pragma omp parallel for schedule(dynamic, 5)
02.   for (i = 0; i < n; i++)
03.   {
04.       unknown_amount_of_work(i);
05.   }
06.
```

Guided scheduling initially assigns initially large chunks of iterations to threads; the number of iterations given to requesting threads is reduced in size as the set of unassigned iterations decreases. Because of the pattern of assignment, guided scheduling tends to require less overhead than dynamic scheduling. The optional chunk argument of the schedule clause denotes the minimum number of iterations in a chunk to be assigned under guided scheduling.

```
- collapse source view plain copy to clipboard print ?
01. #pragma omp parallel for schedule(guided, 8)
02.   for (i = 0; i < n; i++)
03.   {
04.       uneven_amount_of_work(i);
05.   }
06.
```

A special case is when the workload between iterations is monotonically increasing (or decreasing). For example, the number of elements per row in a lower triangular matrix increases in a regular pattern. For such cases, setting a relatively low chunk size (to create a large number of chunks/tasks) with static scheduling may provide an adequate amount of load balance without the overheads needed for dynamic or guided scheduling.

```
- collapse source view plain copy to clipboard print ?
01. #pragma omp parallel for schedule(static, 4)
02.   for (i = 0; i < n; i++)
03.   {
04.       process_lower_triangular_row(i);
05.   }
06.
```

When the choice of schedule is not apparent, use of the runtime schedule allows the alteration of chunk size and schedule type as desired, without requiring recompilation of the program.

When using the `parallel_for` algorithm from Intel® Threading Building Blocks (Intel® TBB), the scheduler divides the iteration space into small tasks that are assigned to threads. If the computation time of some iterations proves to take longer than

other iterations, the Intel TBB scheduler is able to dynamically “steal” tasks from threads in order to achieve a better work load balance among threads.

Explicit threading models (e.g., Windows* threads, Pthreads*, and Java* threads) do not have any means to automatically schedule a set of independent tasks to threads. When needed, such capability must be programmed into the application. Static scheduling of tasks is a straightforward exercise. For dynamic scheduling, two related methods are easily implemented: Producer/Consumer and Boss/Worker. In the former, one thread (Producer) places tasks into a shared queue structure while the Consumer threads remove tasks to be processed, as needed. While not strictly necessary, the Producer/Consumer model is often used when there is some pre-processing to be done before tasks are made available to Consumer threads.

Under the Boss/Worker model, Worker threads rendezvous with the Boss thread whenever more work is needed, to receive assignments directly. In situations where the delineation of a task is very simple, such as a range of indices to an array of data for processing, a global counter with proper synchronization can be used in place of a separate Boss thread. That is, Worker threads access the current value and adjust (likely increment) the counter for the next thread requesting additional work.

Whatever task scheduling model is used, consideration must be given to using the correct number and mix of threads to ensure that threads tasked to perform the required computations are not left idle. For example, if Consumer threads stand idle at times, a reduction in the number of Consumers or an additional Producer thread may be needed. The appropriate solution will depend on algorithmic considerations as well as the number and length of tasks to be assigned.

Usage Guidelines

Any dynamic task scheduling method will entail some overhead as a result of parceling out tasks. Bundling small independent tasks together as a single unit of assignable work can reduce this overhead; correspondingly, if using OpenMP schedule clauses, set a non-default chunk size that will be the minimum number of iterations within a task. The best choice for how much computation constitutes a task will be based on the computation to be done as well as the number of threads and other resources available at execution time.

Additional Resources

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

[Clay Breshears, The Art of Concurrency, O'Reilly Media, Inc., 2009.](#)

[Barbara Chapman, Gabriele Jost, and Ruud van der Post, Using OpenMP: Portable Shared Memory Parallel Programming, The MIT Press, 2007.](#)

[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.](#)

[M. Ben-Ari, Principles of Concurrent and Distributed Programming, Second Edition, Addison-Wesley, 2006.](#)

1.5 – Expose Parallelism by Avoiding or Removing Artificial Dependencies

Abstract

Load balancing an application workload among threads is critical to performance. The key objective for load balancing is to minimize idle time on threads. Sharing the workload equally across all threads with minimal work sharing overheads results in fewer cycles wasted with idle threads not advancing the computation, and thereby leads to improved performance. However, achieving perfect load balance is non-trivial, and it depends on the parallelism within the application, workload, the number of threads, load balancing policy, and the threading implementation.

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:

```
- collapse source  view plain  copy to clipboard  print ?
01.  for each pixel in {imageIn}
02.      sum = value of pixel
03.      // compute the average of 9 pixels from imageIn
04.      for each neighbor of {pixel}
05.          sum += value of neighbor
06.      // store the resulting value in imageOut
07.      pixelOut = sum / 9
08.
```

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:

```
- collapse source  view plain  copy to clipboard  print ?
01.  subroutine BlurLine (lineIn, lineOut)
02.      for each pixel j in {lineIn}
03.          // compute the average of 3 pixels from line
04.          // and store the resulting value in lineout
05.          pixelOut = (pixel j-1 + pixel j + pixel j+1) / 3
06.
07.      declare lineCache[ 3]
08.      lineCache[ 0] = 0
09.      BlurLine (line 1 of imageIn, lineCache[ 1] )
10.      for each line i in {imageIn}
11.          BlurLine (line i+1 of imageIn, lineCache[ i mod 3] )
12.          lineSums = lineCache[ 0] + lineCache[ 1] + lineCache[ 2]
13.          lineOut = lineSums / 3
14.
```

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:

```
- collapse source  view plain  copy to clipboard  print  ?
01. ptr = &someArray[ 0]
02. for (i = 0; i < N; i++)
03. {
04.     Compute (ptr);
05.     ptr++;
06. }
07.
```

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:

```
- collapse source  view plain  copy to clipboard  print  ?
01. // One time operation:
02. // Decompose the image into non-overlapping blocks.
03. blockList = Decompose (image, xRes, yRes)
04.
05. foreach (block in blockList)
06. {
07.     BlurBlock (block, imageIn, imageOut)
08. }
09.
```

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.

```
- collapse source  view plain  copy to clipboard  print  ?
01. ptr = &someArray[ 0]
02. for (i = 0; i < N; i++)
03. {
04.     Compute (ptr[ i ] );
05. }
06.
```

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](#)

1.6 - Using Tasks Instead of Threads

Abstract

Tasks are a lightweight alternative to threads that provide faster startup and shutdown times, better load balancing, an efficient use of available resources, and a higher level of abstraction. Programming models that include task-based programming are Intel® Threading Building Blocks (Intel® TBB), and OpenMP*. This article provides a brief overview of task-based programming and some important guidelines for deciding when to use threads and when to use tasks.

Background

Programming directly with a native threading package is often a poor choice for multithreaded programming. Threads created with these packages are logical threads that are mapped by the operating system onto the physical threads of the hardware. Creating too few logical threads will undersubscribe the system, wasting some of the available hardware resources. Creating too many logical threads will oversubscribe the system, causing the operating system to incur considerable overhead as it must time-slice access to the hardware resources. By using native threads directly, the developer becomes responsible for matching the parallelism available in the application with the resources available in the hardware.

One common way to perform this difficult balancing act is to create a pool of threads that are used across the lifetime of an application. Typically, one logical thread is created per physical thread. The application then dynamically schedules computations on to threads in the thread pool. Using a thread pool not only helps to match parallelism to hardware resources but also avoids the overheads incurred by repeated thread creation and destruction.

Some parallel programming models, such as Intel TBB and the OpenMP API, provide developers with the benefits of thread pools without the burden of explicitly managing the pools. Using these models, developers express the logical parallelism in their applications with tasks, and the runtime library schedules these tasks on to its internal pool of worker threads. Using tasks, developers can focus on the logical parallelism in their application without worrying about managing the parallelism. Also, since tasks are much lighter weight than threads, it is possible to express parallelism at a much finer granularity.

A simple example of using tasks is shown below. The function `fibTBB` calculates the *n*th Fibonacci number using a TBB `task_group`. At each call where *n* >= 10, a task group is created and two tasks are run. In this example, a lambda expression (a feature in the proposed C++0x standard) that describes each task is passed to the function `run`. These calls spawn the tasks, which makes them available for threads in the thread pool to execute. The subsequent call to the function `wait` blocks until all of the tasks in the task group have run to completion.

```

- collapse source  view plain  copy to clipboard  print  ?
01.  int fibTBB(int n) {
02.      if( n<10 ) {
03.          return fibSerial(n);
04.      } else {
05.          int x, y;
06.          tbb::task_group g;
07.          g.run([ &]{ x=fib(n-1); }); // spawn a task
08.          g.run([ &]{ y=fib(n-2); }); // spawn another task
09.          g.wait();                  // wait for both tasks to complete
10.          return x+y;
11.      }
12.  }

```


The routine `fibSerial` is presumed to be a serial variant. Though tasks enable finer grain parallelism than threads, they still have significant overhead compared to a subroutine call. Therefore, it generally pays to solve small subproblems serially.

Another library that supports tasks is the OpenMP API. Unlike Intel TBB, both of these models use compiler support, which makes their interfaces simpler but less portable. For example, the same Fibonacci example shown above using TBB tasks is implemented as `fibOpenMP` below using OpenMP tasks. Because OpenMP requires compiler support, simpler pragmas can be used to denote tasks. However, only compilers that support the OpenMP API will understand these pragmas.

```

- collapse source view plain copy to clipboard print ?
01.  int fibOpenMP( int n ) {
02.      int i, j;
03.      if( n < 10 ) {
04.          return fibSerial(n);
05.      } else {
06.          // spawn a task
07.          #pragma omp task shared( i ), untied
08.          i = fib( n - 1 );
09.          // spawn another task
10.          #pragma omp task shared( j ), untied
11.          j = fib( n - 2 );
12.          // wait for both tasks to complete
13.          #pragma omp taskwait
14.          return i + j;
15.      }
16.  }

```

Intel TBB and the OpenMP API manage task scheduling through work stealing. In work stealing, each thread in the thread pool maintains a local task pool that is organized as a deque (double-ended queue). A thread uses its own task pool like a stack, pushing new tasks that it spawns onto the top of this stack. When a thread finishes executing a task, it first tries to pop a task from the top of its local stack. The task on the top of the stack is the newest and therefore most likely to access data that is hot in its data cache. If there are no tasks in its local task pool, however, it attempts to steal work from another thread (the victim). When stealing, a thread uses the victim's deque like a queue so that it steals the oldest task from the victim's deque. For recursive algorithms, these older tasks are nodes that are high in the task tree and therefore are large chunks of work, often work that is not hot in the victim's data cache. Therefore, work stealing is an effective mechanism for balancing load while maintaining cache locality.

The thread pool and the work-stealing scheduler that distributes work across the threads are hidden from developers when a tasking library is used. Therefore, tasks provide a high-level abstraction that lets users think about the logical parallelism in their application without worrying about managing the parallelism. The load balancing provided by work-stealing and the low creation and destruction costs for tasks make task-based parallelism an effective solution for most applications.

Usage Guidelines

While using tasks is usually the best approach to adding threading for performance, there are cases when using tasks is not appropriate. The task schedulers used by Intel TBB and the OpenMP API are non-preemptive. Tasks are therefore intended for high-performance algorithms that are non-blocking. They still work well if the tasks rarely block. However, if tasks block frequently, there is a performance loss because while a task is blocked, the thread it has been assigned to cannot work on any other tasks. Blocking typically occurs while waiting for I/O or mutexes for long periods. If threads hold mutexes for long periods, the code is not likely to perform well, regardless of how many threads it has. For blocking tasks, it is best to use threads rather than tasks.

Even in cases when using tasks is best, sometimes it's not necessary to implement the tasking pattern from scratch. The Intel TBB library provides not only a task interface but also high-level algorithms that implement some of the most common task patterns, such as `parallel_invoke`, `parallel_for`, `parallel_reduce` and `pipeline`. The OpenMP API offers parallel loops. Since these patterns have been tuned and tested, it's best to use these high-level algorithms whenever possible.

The example below shows a simple serial loop and a parallel version of the loop that uses the `tbb::parallel_for` algorithm:

```
- collapse source  view plain  copy to clipboard  print  ?
01.  // serial loop
02.  for (int i = 0; i < 10000; ++i)
03.      a[i] = f(i) + g(i);
04.
05.  // parallel loop
06.  tbb::parallel_for( 0, 10000, [&](int i) { a[i] = f(i) + g(i); } );
07.
```

In the example above, the TBB `parallel_for` creates tasks that apply the loop body, in this case `a[i] = f(i) + g(i)`, to each of the elements in the range `[0,10000)`. The `&` in the lambda expression indicates that variable `a` should be captured by reference. When using a `parallel_for`, the TBB runtime library chooses an appropriate number of iterations to group together in a single task to minimize overheads while providing ample tasks for load balancing.

Additional Resources

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

[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.](#)

1.7 – Exploiting Data Parallelism in Ordered Data Streams

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.

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 is 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:

```

- collapse source  view plain  copy to clipboard  print  ?
01.  inFile = OpenFile ()
02.  outFile == InitializeOutputFile ()
03.  WriteHeader (outFile)
04.  outputBuffer = AllocateBuffer (bufferSize)
05.  while (frame = ReadNextFrame (inFile))
06.  {
07.      EncodeFrame (frame, outputBuffer)
08.      if (outputBuffer size > bufferThreshold)
09.          FlushBuffer (outputBuffer, outFile)
10.  }
11.  FlushBuffer (outputBuffer, outFile)
12.

```

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:

```

- collapse source  view plain  copy to clipboard  print  ?
01.  WriteHeader (outFile)
02.  while (block = ReadNextBlock (inFile))
03.  {
04.      while (frame = ReadNextFrame (block))
05.      {
06.          EncodeFrame (frame, outputBuffer)
07.          if (outputBuffer size > bufferThreshold)
08.              FlushBuffer (outputBuffer, outFile)
09.      }
10.      FlushBuffer (outputBuffer, outFile)
11.  }
12.

```

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:

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:

```

- collapse source  view plain  copy to clipboard  print  ?
01.  while (block = ReadNextBlock (inFile))
02.  {
03.      WriteHeader (outputBuffer)
04.      while (frame = ReadNextFrame (block))
05.      {
06.          EncodeFrame (frame, outputBuffer)
07.      }
08.      FlushBuffer (outputBuffer, outFile)
09.  }
10.

```

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

```

- collapse source  view plain  copy to clipboard  print  ?
01.  // Create a team of threads with private
02.  // copies of outputBuffer, block, and frame
03.  // and shared copies of inFile and outFile
04.  while (AcquireLock,
05.         block = ReadNextBlock (inFile),
06.         ReleaseLock, block)
07.  {
08.      WriteHeader (outputBuffer)
09.      while (frame = ReadNextFrame (block))
10.      {
11.          EncodeFrame (frame, outputBuffer)
12.      }
13.      FlushBuffer (outputBuffer, outFile)
14.  }
15.
16.

```

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.

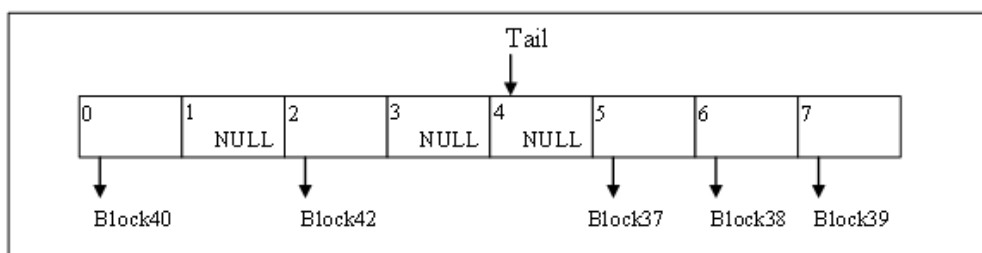


Figure 1. State of example reorder buffer before writing.

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.

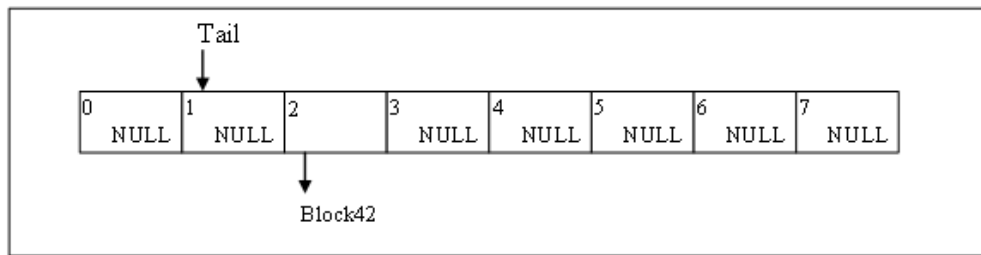


Figure 2. State of example reorder buffer after writing.

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:

```

- collapse source  view plain  copy to clipboard  print  ?
01.  inFile = OpenFile ()
02.  outFile == InitializeOutputFile ()
03.  // Create a team of threads with private
04.  // copies of outputBuffer, block, and frame, shared
05.  // copies of inFile and outFile.
06.  while (AcquireLock,
07.         block = ReadNextBlock (inFile),
08.         ReleaseLock, block)
09.  {
10.    WriteHeader (outputBuffer)
11.    while (frame = ReadNextFrame (block))
12.    {
13.      EncodeFrame (frame, outputBuffer)
14.    }
15.    QueueOrFlush (outputBuffer, outFile)
16.  }
17.

```

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](#)

2.1 – Managing Lock Contention: Large and Small Critical Sections

Abstract

In multithreaded applications, locks are used to synchronize entry to regions of code that access shared resources. The region of code protected by these locks is called a critical section. While one thread is inside a critical section, no other thread can enter. Therefore, critical sections serialize execution. This topic introduces the concept of critical section size, defined as the length of time a thread spends inside a critical section, and its effect on performance.

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

Background

Critical sections ensure data integrity when multiple threads attempt to access shared resources. They also serialize the execution of code within the critical section. Threads should spend as little time inside a critical section as possible to reduce the amount of time other threads sit idle waiting to acquire the lock, a state known as “lock contention.” In other words, it is best to keep critical sections small. However, using a multitude of small, separate critical sections introduces system overheads associated with acquiring and releasing each separate lock. This article presents scenarios that illustrate when it is best to use large or small critical sections.

The thread function in Code Sample 1 contains two critical sections. Assume that the critical sections protect different data and that the work in functions DoFunc1 and DoFunc2 is independent. Also assume that the amount of time to perform either of the update functions is always very small.

Code Sample 1:

```

- collapse source  view plain  copy to clipboard  print  ?
01.  Begin Thread Function ()
02.      Initialize ()
03.
04.      BEGIN CRITICAL SECTION 1
05.          UpdateSharedData1 ()
06.      END CRITICAL SECTION 1
07.
08.      DoFunc1 ()
09.
10.      BEGIN CRITICAL SECTION 2
11.          UpdateSharedData2 ()
12.      END CRITICAL SECTION 2
13.
14.      DoFunc2 ()
15.  End Thread Function ()
16.
17.

```

The critical sections are separated by a call to DoFunc1. If the threads only spend a small amount of time in DoFunc1, the synchronization overhead of two critical sections may not be justified. In this case, a better scheme might be to merge the two small critical sections into one larger critical section, as in Code Sample 2.

Code Sample 2:

```

- collapse source  view plain  copy to clipboard  print  ?
01.  Begin Thread Function {}
02.      Initialize {}
03.
04.      BEGIN CRITICAL SECTION 1
05.          UpdateSharedData1 {}
06.          DoFunc1 {}
07.          UpdateSharedData2 {}
08.      END CRITICAL SECTION 1
09.
10.      DoFunc2 {}
11.  End Thread Function {}
12.

```

If the time spent in `DoFunc1` is much higher than the combined time to execute both update routines, this might not be a viable option. The increased size of the critical section increases the likelihood of lock contention, especially as the number of threads increases.

Consider a variation of the previous example where the threads spend a large amount of time in the `UpdateSharedData2` function. Using a single critical section to synchronization access to `UpdateSharedData1` and `UpdateSharedData2`, as in Code Sample 2, is no longer a good solution because the likelihood of lock contention is higher. On execution, the thread that gains access to the critical section spends a considerable amount of time in the critical section, while all the remaining threads are blocked. When the thread holding the lock releases it, one of the waiting threads is allowed to enter the critical section and all other waiting threads remain blocked for a long time. Therefore, Code Sample 1 is a better solution for this case.

It is good programming practice to associate locks with particular shared data. Protecting all accesses of a shared variable with the same lock does not prevent other threads from accessing a different shared variable protected by a different lock. Consider a shared data structure. A separate lock could be created for each element of the structure, or a single lock could be created to protect access to the whole structure. Depending on the computational cost of updating the elements, either of these extremes could be a practical solution. The best lock granularity might also lie somewhere in the middle. For example, given a shared array, a pair of locks could be used: one to protect the even numbered elements and the other to protect the odd numbered elements.

In the case where `UpdateSharedData2` requires a substantial amount of time to complete, dividing the work in this routine and creating new critical sections may be a better option. In Code Sample 3, the original `UpdateSharedData2` has been broken up into two functions that operate on different data. It is hoped that using separate critical sections will reduce lock contention. If the entire execution of `UpdateSharedData2` did not need protection, rather than enclose the function call, critical sections inserted into the function at points where shared data are accessed should be considered.

Code Sample 3:

```

- collapse source  view plain  copy to clipboard  print  ?
01.  Begin Thread Function ()
02.      Initialize ()
03.
04.      BEGIN CRITICAL SECTION 1
05.          UpdateSharedData1 ()
06.      END CRITICAL SECTION 1
07.
08.      DoFunc1 ()
09.
10.      BEGIN CRITICAL SECTION 2
11.          UpdateSharedData2 ()
12.      END CRITICAL SECTION 2
13.
14.      BEGIN CRITICAL SECTION 3
15.          UpdateSharedData3 ()
16.      END CRITICAL SECTION 3
17.
18.      DoFunc2 ()
19.  End Thread Function ()
20.

```

Advice

Balance the size of critical sections against the overhead of acquiring and releasing locks. Consider aggregating small critical sections to amortize locking overhead. Divide large critical sections with significant lock contention into smaller critical sections. Associate locks with particular shared data such that lock contention is minimized. The optimum solution probably lies somewhere between the extremes of a different lock for every shared data element and a single lock for all shared data.

Remember that synchronization serializes execution. Large critical sections indicate that the algorithm has little natural concurrency or that data partitioning among threads is sub-optimal. Nothing can be done about the former except changing the algorithm. For the latter, try to create local copies of shared data that the threads can access asynchronously.

The forgoing discussion of critical section size and lock granularity does not take the cost of context switching into account. When a thread blocks at a critical section waiting to acquire the lock, the operating system swaps an active thread for the idle thread. This is known as a context switch. In general, this is the desired behavior, because it releases the CPU to do useful work. For a thread waiting to enter a small critical section, however, a spin-wait may be more efficient than a context switch. The waiting thread does not relinquish the CPU when spin-waiting. Therefore, a spin-wait is only recommended when the time spent in a critical section is less than the cost of a context switch.

Code Sample 4 shows a useful heuristic to employ when using the Win32 threading API. This example uses the spin-wait option on Win32 CRITICAL_SECTION objects. A thread that is unable to enter a critical section will spin rather than relinquish the CPU. If the CRITICAL_SECTION becomes available during the spin-wait, a context switch is avoided. The spin-count parameter determines how many times the thread will spin before blocking. On uniprocessor systems, the spin-count parameter is ignored. Code Sample 4 uses a spin-count of 1000 for each thread in the application but a maximum spin-count of 8000.

Code Sample 4:

```

- collapse source  view plain  copy to clipboard  print  ?
01.  int gNumThreads;
02.  CRITICAL_SECTION gCs;
03.
04.  int main ()
05.  {
06.      int spinCount = 0;
07.      ...
08.      spinCount = gNumThreads * 1000;
09.      if (spinCount > 8000) spinCount = 8000;
10.      InitializeCriticalSectionAndSpinCount (&gCs, spinCount);
11.      ...
12.  }
13.
14.  DWORD WINAPI ThreadFunc (void *data)
15.  {
16.      ...
17.      EnterCriticalSection (&gCs);
18.      ...
19.      LeaveCriticalSection (&gCs);
20.  }
21.

```

Usage Guidelines

The spin-count parameter used in Code Sample 4 should be tuned differently on processors with Intel® Hyper-Threading Technology (Intel® HT Technology), where spin-waits are generally detrimental to performance. Unlike true symmetric multiprocessor (SMP) systems, which have multiple physical CPUs, Intel HT Technology creates two logical processors on the same CPU core. Spinning threads and threads doing useful work must compete for logical processors. Thus, spinning threads can impact the performance of multithreaded applications to a greater extent on systems with Intel HT Technology compared to SMP systems. The spin-count for the heuristic in Code Sample 4 should be lower or not used at all.

Additional Resources

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

[Intel® Hyper-Threading Technology](#)

2.2 – Use Synchronization Routines Provided by the Threading API Rather than Hand-Coded Synchronization

Abstract

Application programmers sometimes write hand-coded synchronization routines rather than using constructs provided by a threading API in order to reduce synchronization overhead or provide different functionality than existing constructs offer. Unfortunately, using hand-coded synchronization routines may have a negative impact on performance, performance tuning, or debugging of multi-threaded applications.

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

It is often tempting to write hand-coded synchronization to avoid the overhead sometimes associated with the synchronization routines from the threading API. Another reason programmers write their own synchronization routines is that those provided by the threading API do not exactly match the desired functionality. Unfortunately, there are serious disadvantages to hand-coded synchronization compared to using the threading API routines.

One disadvantage of writing hand-coded synchronization is that it is difficult to guarantee good performance across different hardware architectures and operating systems. The following example is a hand-coded spin lock written in C that will help illustrate these problems:

```
- collapse source  view plain  copy to clipboard  print  ?
01.  #include <ia64intrin.h>
02.
03.  void acquire_lock (int *lock)
04.  {
05.      while (!_InterlockedCompareExchange (lock, TRUE, FALSE) == TRUE);
06.  }
07.
08.  void release_lock (int *lock)
09.  {
10.      *lock = FALSE;
11.  }
12.  </ia64intrin.h>
```

The `_InterlockedCompareExchange` compiler intrinsic is an interlocked memory operation that guarantees no other thread can modify the specified memory location during its execution. It first compares the memory contents of the address in the first argument with the value in the third argument, and if a match occurs, stores the value in the second argument to the memory address specified in the first argument. The original value found in the memory contents of the specified address is returned by the intrinsic. In this example, the `acquire_lock` routine spins until the contents of the memory location `lock` are in the unlocked state (`FALSE`) at which time the lock is acquired (by setting the contents of `lock` to `TRUE`) and the routine returns. The `release_lock` routine sets the contents of the memory location `lock` back to `FALSE` to release the lock.

Although this lock implementation may appear simple and reasonably efficient at first glance, it has several problems:

- If many threads are spinning on the same memory location, cache invalidations and memory traffic can become excessive at the point when the lock is released, resulting in poor scalability as the number of threads increases.
- This code uses an atomic memory primitive that may not be available on all processor architectures, limiting portability.
- The tight spin loop may result in poor performance for certain processor architecture features, such as Intel® Hyper-Threading Technology.
- The `while` loop appears to the operating system to be doing useful computation, which could negatively impact the fairness of operating system scheduling.

Although techniques exist for solving all these problems, they often complicate the code enormously, making it difficult to verify correctness. Also, tuning the code while maintaining portability can be difficult. These problems are better left to the authors of the threading API, who have more time to spend verifying and tuning the synchronization constructs to be portable and scalable.

Another serious disadvantage of hand-coded synchronization is that it often decreases the accuracy of programming tools for threaded environments. For example, the Intel® Parallel Studio tools need to be able to identify synchronization constructs in order to provide accurate information about performance (using Intel® Parallel Amplifier) and correctness (using Intel® Parallel Inspector) of the threaded application program.

Threading tools are often designed to identify and characterize the functionality of the synchronization constructs provided by the supported threading API(s). Synchronization is difficult for the tools to identify and understand if standard synchronization APIs are not used to implement it, which is the case in the example above.

Sometimes tools support hints from the programmer in the form of tool-specific directives, pragmas, or API calls to identify and characterize hand-coded synchronization. Such hints, even if they are supported by a particular tool, may result in less accurate analysis of the application program than if threading API synchronization were used: the reasons for performance problems may be difficult to detect or threading correctness tools may report spurious race conditions or missing synchronization.

Advice

Avoid the use of hand-coded synchronization if possible. Instead, use the routines provided by your preferred threading API, such as `queueing_mutex` or `spin_mutex` for Intel® Threading Building Blocks, `omp_set_lock/omp_unset_lock` or `critical/end critical` directives for OpenMP*, or `pthread_mutex_lock/pthread_mutex_unlock` for Pthreads*. Study the threading API synchronization routines and constructs to find one that is appropriate for your application.

If a synchronization construct is not available that provides the needed functionality in the threading API, consider using a different algorithm for the program that requires less or different synchronization. Furthermore, expert programmers could build a custom synchronization construct from simpler synchronization API constructs instead of starting from scratch. If hand-coded synchronization must be used for performance reasons, consider using pre-processing directives to enable easy replacement of the hand-coded synchronization with a functionally equivalent synchronization from the threading API, thus increasing the accuracy of the threading tools.

Usage Guidelines

Programmers who build custom synchronization constructs from simpler synchronization API constructs should avoid using spin loops on shared locations to avoid non-scalable performance. If the code must also be portable, avoiding the use of atomic memory primitives is also advisable. The accuracy of threading performance and correctness tools may suffer because the tools may not be able to deduce the functionality of the custom synchronization construct, even though the simpler synchronization constructs from which it is built may be correctly identified.

Additional Resources

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

[John Mellor-Crummey, "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors." ACM Transactions on Computer Systems, Vol. 9, No. 1, February 1991. Pages 21-65.](#)

[Intel Pentium 4 and Intel Xeon Processor Optimization Reference Manual, Chapter 7: "Multiprocessor and Hyper-Threading Technology." Order Number: 248966-007.](#)

[Intel Parallel Studio](#)

[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.](#)

2.3 – Choosing Appropriate Synchronization Primitives to Minimize Overhead

Abstract

When threads wait at a synchronization point, they are not doing useful work. Unfortunately, some degree of synchronization is usually necessary in multithreaded programs, and explicit synchronization is sometimes even preferred over data duplication or complex non-blocking scheduling algorithms, which in turn have their own problems. Currently, there are a number of synchronization mechanisms available, and it is left to the application developer to choose an appropriate one to minimize overall synchronization overhead.

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

Background

By nature, synchronization constructs serialize execution, limiting parallelism and potentially decreasing overall application performance. In reality, however, very few multithreaded programs are entirely synchronization-free. Fortunately, it is possible to mitigate some of the system overhead associated with synchronization by choosing appropriate constructs. This article reviews several available approaches, provides sample code for each, and lists key advantages and disadvantages.

Win32* Synchronization API

The Win32 API provides several mechanisms to guarantee atomicity, three of which are discussed in this section. An increment statement (e.g., `var++`) illustrates the different constructs. If the variable being updated is shared among threads, the load>write>store instructions must be atomic (i.e., the sequence of instructions must not be preempted before completion). The code below demonstrates how these APIs can be used:

```
- collapse source  view plain  copy to clipboard  print  ?
01.  #include <windows.h>
02.
03.  CRITICAL_SECTION cs; /* InitializeCriticalSection called in main() */
04.  HANDLE mtx; /* CreateMutex called in main() */
05.  static LONG counter = 0;
06.
07.  void IncrementCounter ()
08.  {
09.      // Synchronize with Win32 interlocked function
10.      InterlockedIncrement (&counter);
11.
12.      // Synchronize with Win32 critical section
13.      EnterCriticalSection (&cs);
14.      counter++;
15.      LeaveCriticalSection (&cs);
16.
17.      // Synchronize with Win32 mutex
18.      WaitForSingleObject (mtx, INFINITE);
19.      counter++;
20.      ReleaseMutex (mtx);
21.  }
22.  </windows.h>
```

Compare these three mechanisms to illustrate which one could be more suitable in various synchronization scenarios. The Win32 interlocked functions (`InterlockedIncrement`, `InterlockedDecrement`, `InterlockedExchange`, `InterlockedExchangeAdd`, `InterlockedCompareExchange`) are limited to simple operations, but they are faster than critical regions. In addition, fewer function calls are required; to enter and exit a Win32 critical region requires calls to `EnterCriticalSection` and `LeaveCriticalSection` or `WaitForSingleObject` and `ReleaseMutex`. The interlocked functions are also non-blocking, whereas `EnterCriticalSection` and `WaitForSingleObject` (or `WaitForMultipleObjects`) block threads if the synchronization object is not available.

When a critical region is necessary, synchronizing on a Win32 `CRITICAL_SECTION` requires significantly less system overhead than Win32 mutex, semaphore, and event `HANDLE`s because the former is a user-space object whereas the latter are kernel-space objects. Although Win32 critical sections are usually faster than Win32 mutexes, they are not as versatile. Mutexes, like other kernel objects, can be used for inter-process synchronization. Timed-waits are also possible with the `WaitForSingleObject` and `WaitForMultipleObjects` functions. Rather than wait indefinitely to acquire a mutex, the threads continue after the specified time limit expires. Setting the wait-time to zero allows threads to test whether a mutex is available without blocking. (Note that it is also possible to check the availability of a `CRITICAL_SECTION` without blocking using the `TryEnterCriticalSection` function.) Finally, if a thread terminates while holding a mutex, the operating system signals the handle to prevent waiting threads from becoming deadlocked. If a thread terminates while holding a `CRITICAL_SECTION`, threads waiting to enter this `CRITICAL_SECTION` are deadlocked.

A Win32 thread immediately relinquishes the CPU to the operating system when it tries to acquire a `CRITICAL_SECTION` or mutex `HANDLE` that is already held by another thread. In general, this is good behavior. The thread is blocked and the CPU is free to do useful work. However, blocking and unblocking a thread is expensive. Sometimes it is better for the thread to try to acquire the lock again before blocking (e.g., on SMP systems, at small critical sections). Win32 `CRITICAL_SECTION`s have a user-configurable spin-count to control how long threads should wait before relinquishing the CPU. The `InitializeCriticalSectionAndSpinCount` and `SetCriticalSectionSpinCount` functions set the spin-count for threads trying to enter a particular `CRITICAL_SECTION`.

Advice

For simple operations on variables (i.e., increment, decrement, exchange), use fast, low-overhead Win32 interlocked functions.

Use Win32 mutex, semaphore, or event `HANDLE`s when inter-process synchronization or timed-waits are required. Otherwise, use Win32 `CRITICAL_SECTION`s, which have lower system overhead.

Control the spin-count of Win32 `CRITICAL_SECTION`s using the `InitializeCriticalSectionAndSpinCount` and `SetCriticalSectionSpinCount` functions. Controlling how long a waiting thread spins before relinquishing the CPU is especially important for small and high-contention critical sections. Spin-count can significantly impact performance on SMP systems and processors with Intel® Hyper-Threading Technology.

Intel® Threading Building Blocks Synchronization API

Intel® Threading Building Blocks (Intel® TBB) provides portable wrappers around atomic operations (template class `atomic<T>`) and mutual exclusion mechanisms of different flavors, including a wrapper around a “native” mutex. Since the previous section

already discussed advantages and disadvantages of using atomic operations and operating system-dependent synchronization API, this section skips `tbb::atomic<T>` and `tbb::mutex`, focusing instead on fast user-level synchronization classes, such as `spin_mutex`, `queuing_mutex`, `spin_rw_mutex`, and `queuing_rw_mutex`.

The simplest mutex is `spin_mutex`. A thread trying to acquire a lock on a `spin_mutex` busy waits until it can acquire the lock. A `spin_mutex` is appropriate when the lock is held for only a few instructions. For example, the following code uses a mutex `FreeListMutex` to protect a shared variable `FreeList`:

```
- collapse source  view plain  copy to clipboard  print  ?
01. Node* FreeList;
02. typedef spin_mutex FreeListMutexType;
03. FreeListMutexType FreeListMutex;
04.
05. Node* AllocateNode()
06. {
07.     Node* n;
08.     {
09.         FreeListMutexType::scoped_lock lock(FreeListMutex);
10.         n = FreeList;
11.         if( n )
12.             FreeList = n->next;
13.     }
14.     if( !n )
15.         n = new Node();
16.     return n;
17. }
18.
```

The constructor for `scoped_lock` waits until there are no other locks on `FreeListMutex`. The destructor releases the lock. The role of additional braces inside the `AllocateNode` function is to keep the lifetime of the lock as short as possible, so that other waiting threads can get their chance as soon as possible.

Another user-level spinning mutex provided by Intel TBB is `queuing_mutex`. This also is a user-level mutex, but as opposed to `spin_mutex`, `queuing_mutex` is fair. A fair mutex lets threads through in the order they arrived. Fair mutexes avoid starving threads, since each thread gets its turn. Unfair mutexes can be faster than fair ones, because they let threads that are running go through first, instead of the thread that is next in line, which may be sleeping because of an interrupt. Queuing mutex should be used when mutex scalability and fairness is really important.

Not all accesses to shared data need to be mutually exclusive. In most real-world applications, accesses to concurrent data structures are more often read-accesses and only a few of them are write-accesses. For such data structures, protecting one reader from another one is not necessary, and this serialization can be avoided. Intel TBB reader/writer locks allow multiple readers to enter a critical region, and only a writer thread gets an exclusive access. An unfair version of busy-wait reader/writer mutex is `spin_rw_mutex`, and its fair counterpart is `queuing_rw_mutex`. Reader/writer mutexes supply the same `scoped_lock` API as `spin_mutex` and `queuing_mutex`, and in addition provide special functions to allow a reader lock to upgrade to a writer one and a writer lock to downgrade to a reader lock.

Advice

A key to successful choice of appropriate synchronization is knowing your application, including the data being processed and the processing patterns.

If the critical region is only a few instructions long and fairness is not an issue, then `spin_mutex` should be preferred. In situations where the critical region is fairly short, but it's important to allow threads access to it in the order they arrived to critical region, use `queuing_mutex`.

If the majority of concurrent data accesses are read-accesses and only a few threads rarely require write access to the data, it's possible that using reader/writer locks will help avoid unnecessary serialization and will improve overall application performance.

Usage Guidelines

Beware of thread preemption when making successive calls to Win32 interlocked functions. For example, the following code segments will not always yield the same value for `localVar` when executed with multiple threads:

```
- collapse source  view plain  copy to clipboard  print  ?
01.  static LONG N = 0;
02.  LONG localVar;
03.  ...
04.  InterlockedIncrement (&N);
05.  InterlockedIncrement (&N);
06.  InterlockedExchange (&localVar, N);
07.
08.  static LONG N = 0;
09.  LONG localVar;
10.  ...
11.  EnterCriticalSection (&lock);
12.     localVar = (N += 2);
13.  LeaveCriticalSection (&lock);
14.
```

In the example using interlocked functions, thread preemption between any of the function calls can produce unexpected results. The critical section example is safe because both atomic operations (i.e., the update of global variable `N` and assignment to `localVar`) are protected.

For safety, Win32 critical regions, whether built with `CRITICAL_SECTION` variables or mutex `HANDLE`s, should have only one point of entry and exit. Jumping into critical sections defeats synchronization. Jumping out of a critical section without calling `LeaveCriticalSection` or `ReleaseMutex` will deadlock waiting threads. Single entry and exit points also make for clearer code.

Prevent situations where threads terminate while holding `CRITICAL_SECTION` variables, because this will deadlock waiting threads.

Additional Resources

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

[Johnson M. Hart, Win32 System Programming \(2nd Edition\). Addison-Wesley, 2001](#)

[Jim Beveridge and Robert Wiener, Multithreading Applications in Win32. Addison-Wesley, 1997.](#)

2.4 – Use Non-blocking Locks When Possible

Abstract

Threads synchronize on shared resources by executing synchronization primitives offered by the supporting threading implementation. These primitives (such as mutex, semaphore, etc.) allow a single thread to own the lock, while the other threads either spin or block depending on their timeout mechanism. Blocking results in costly context-switch, whereas spinning results in wasteful use of CPU execution resources (unless used for very short duration). Non-blocking system calls, on the other hand, allow the competing thread to return on an unsuccessful attempt to the lock, and allow useful work to be done, thereby avoiding wasteful utilization of execution resources at the same time.

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

Background

Most threading implementations, including the Windows* and POSIX* threads APIs, provide both blocking and non-blocking thread synchronization primitives. The blocking primitives are often used as default. When the lock attempt is successful, the thread gains control of the lock and executes the code in the critical section. However, in the case of an unsuccessful attempt, a context-switch occurs and the thread is placed in a queue of waiting threads. A context-switch is costly and should be avoided for the following reasons:

- Context-switch overheads are considerable, especially if the threads implementation is based on kernel threads.
- Any useful work in the application following the synchronization call needs to wait for execution until the thread gains control of the lock.

Using non-blocking system calls can alleviate the performance penalties. In this case, the application thread resumes execution following an unsuccessful attempt to lock the critical section. This avoids context-switch overheads, as well as avoidable spinning on the lock. Instead, the thread performs useful work before the next attempt to gain control of the lock.

Advice

Use non-blocking threading calls to avoid context-switch overheads. The non-blocking synchronization calls usually start with the try keyword. For instance, the blocking and non-blocking versions of the critical section synchronization primitive offered by the Windows threading implementation are as follows:

If the lock attempt to gain ownership of the critical section is successful, the `TryEnterCriticalSection` call returns the Boolean value of `True`. Otherwise, it returns `False`, and the thread can continue execution of application code.

```
void EnterCriticalSection (LPCRITICAL_SECTION cs);  
bool TryEnterCriticalSection (LPCRITICAL_SECTION cs);
```

Typical use of the non-blocking system call is as follows:

```

- collapse source  view plain  copy to clipboard  print  ?
01.  CRITICAL_SECTION cs;
02.  void threadfoo()
03.  {
04.      while(TryEnterCriticalSection(&cs) == FALSE)
05.      {
06.          // some useful work
07.      }
08.      // Critical Section of Code
09.      LeaveCriticalSection (&cs);
10.  }
11.  // other work
12.  }
13.
14.

```

Similarly, the POSIX threads provide non-blocking versions of the mutex, semaphore, and condition variable synchronization primitives. For instance, the blocking and non-blocking versions of the mutex synchronization primitive are as follows:

```

int pthread_mutex_lock (pthread_mutex_t *mutex);
int pthread_mutex_try_lock (pthread_mutex_t *mutex);

```

It is also possible to specify timeouts for thread locking primitives in the Windows* threads implementation. The Win32* API provides the `WaitForSingleObject` and `WaitForMultipleObjects` system calls to synchronize on kernel objects. The thread executing these calls waits until the relevant kernel object is signaled or a user specified time interval has passed. Once the timeout interval elapses, the thread can resume executing useful work.

```

DWORD WaitForSingleObject (HANDLE hHandle, DWORD dwMilliseconds);

```

In the code above, `hHandle` is the handle to the kernel object, and `dwMilliseconds` is the timeout interval after which the function returns if the kernel object is not signaled. A value of `INFINITE` indicates that the thread waits indefinitely. A code snippet demonstrating the use of this API call is included below.

```

- collapse source  view plain  copy to clipboard  print  ?
01.  void threadfoo ()
02.  {
03.      DWORD ret_value;
04.      HANDLE hHandle;
05.      // Some work
06.      ret_value = WaitForSingleObject (hHandle,0);
07.
08.      if (ret_value == WAIT_TIME_OUT)
09.      {
10.          // Thread could not gain ownership of the kernel
11.          // object within the time interval;
12.          // Some useful work
13.      }
14.      else if (ret_value == WAIT_OBJECT_0)
15.      {
16.          // Critical Section of Code
17.      }
18.      else { // Handle Wait Failure)
19.          // Some work
20.      }
21.

```

Similarly, the `WaitForMultipleObjects` API call allows the thread to wait on the signal status of multiple kernel objects.

When using a non-blocking synchronization call, for instance, `TryEnterCriticalSection`, verify the return value of the synchronization call to see if the request has been successful before releasing the shared object.

Usage Guidelines

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

[Aaron Cohen and Mike Woodring. Win32 Multithreaded Programming. O'Reilly Media; 1 edition. 1997.](#)

[Jim Beveridge and Robert Wiener. Multithreading Applications in Win32 – the Complete Guide to Threads. Addison-Wesley Professional. 1996.](#)

[Bil Lewis and Daniel J Berg. Multithreaded Programming with Pthreads. Prentice Hall PTR; 136th edition. 1997.](#)

Authors and Editors

The following Intel engineers and technical experts contributed to writing, reviewing and editing the Intel® Guide for Developing Multithreaded Applications: Henry Gabb, Martyn Corden, Todd Rosenquist, Paul Fischer, Julia Fedorova, Clay Breshears, Thomas Zipplies, Vladimir Tsymbal, Levent Akyil, Anton Pegushin, Alexey Kukanov, Paul Petersen, Mike Voss, Aaron Tersteeg and Jay Hoeflinger.

Learn more about developer tools and resources at [Intel® Software Development Products](#)

Brought to you by Intel® Software Dispatch
Delivering the latest tools, techniques, and best practices for leading-edge software infrastructure, platform software, and developer resources.

