Granularity and Parallel Performance

<keywords: threading, performance, granularity, OpenMP, synchronization>

Abstract

One key to attaining 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.

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

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 it is 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. Therefore, 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*:

```
#pragma omp parallel
{ int j, limit, prime;
#pragma for schedule(dynamic, 1)
  for(i = 3; i <= 1000000; i += 2) {
    limit = (int) sqrt((float)i) + 1;
    prime = 1; // assume number is prime
    j = 3;
    while (prime && (j <= limit)) {
      if (i%j == 0) prime = 0;
```

```
        j += 2;
      }

      if (prime) {
        #pragma omp critical
        {
        numPrimes++;
        if (i%4 == 1) numP41++;  // 4k+1 primes
        if (i%4 == 3) numP43++;  // 4k-1 primes
        }
      }
    }
  }
```

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.

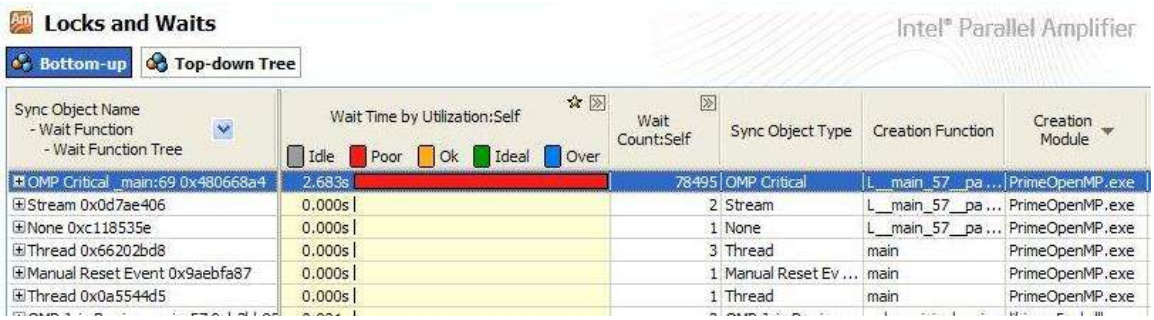

Figure 1. Locks and Waits analysis results demonstrating that the OpenMP* critical region is cause of synchronization overhead.

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:

```
#pragma omp parallel
{
  int j, limit, prime;

  #pragma for schedule(dynamic, 1) \
    reduction(+:numPrimes,numP41,numP43)
  for(i = 3; i <= 1000000; i += 2) {
    limit = (int) sqrt((float)i) + 1;
    prime = 1;  // assume number is prime
    j = 3;
    while (prime && (j <= limit))
    {
      if (i%j == 0) prime = 0;
      j += 2;
    }

    if (prime)
```

```
      {
        numPrimes++;
        if (i%4 == 1) numP41++;  // 4k+1 primes
        if (i%4 == 3) numP43++;  // 4k-1 primes
      }
    }
  }
```

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:

```
#pragma omp parallel
{
  int j, limit, prime;
  #pragma for schedule(dynamic, 100000) \
    reduction(+:numPrimes, numP41, numP43)
  for(i = 3; i <= 1000000; i += 2)
  {
    limit = (int) sqrt((float)i) + 1;
    prime = 1; // assume number is prime
    j = 3;
    while (prime && (j <= limit))
    {
      if (i%j == 0) prime = 0;
      j += 2;
    }

    if (prime)
    {
      numPrimes++;
      if (i%4 == 1) numP41++;  // 4k+1 primes
      if (i%4 == 3) numP43++;  // 4k-1 primes
    }
  }
}
```

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.

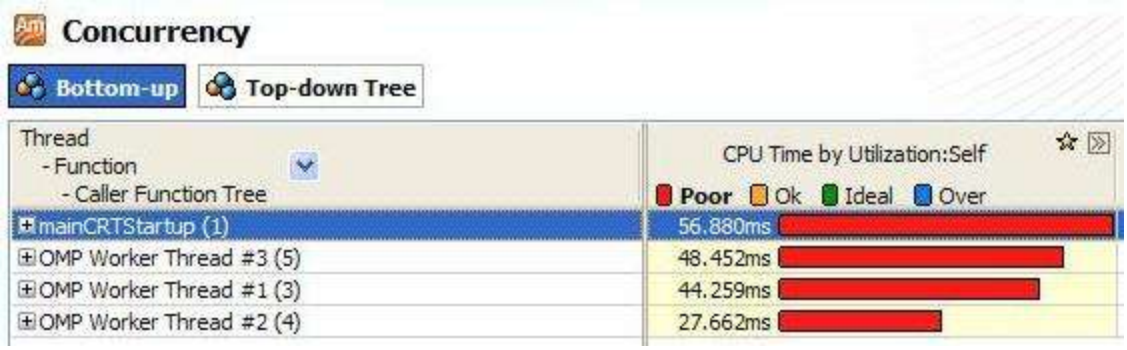

Figure 2. Concurrency analysis results demonstrating the imbalance of execution time used by each thread.

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.

```
#pragma omp parallel
{
  int j, limit, prime;
  #pragma for schedule(static, 100) \
    reduction(+:numPrimes, numP41, numP43)
  for(i = 3; i <= 1000000; i += 2)
  {
    limit = (int) sqrt((float)i) + 1;
    prime = 1;  // assume number is prime
    j = 3;
    while (prime && (j <= limit))
    {
      if (i%j == 0) prime = 0;
      j += 2;
    }

    if (prime)
    {
      numPrimes++;
      if (i%4 == 1) numP41++;  // 4k+1 primes
      if (i%4 == 3) numP43++;  // 4k-1 primes
    }
  }
}
```

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.