Curing Thread Imbalance Using Intel® Parallel Amplifier

<keywords: threading, scalability, Parallel Amplifier, scheduling, hotspot, utilization, concurrency>

Abstract

One of the performance-inhibiting factors in threaded applications is load imbalance. Balancing the workload among threads is critical to application performance. The key objective for load balancing is to minimize idle time on threads and share the workload equally across all threads with minimal work sharing overheads. Intel® Parallel Amplifier, an Intel® Parallel Studio product, assists in fine-tuning parallel applications for optimal performance on multicore processors. Intel Parallel Amplifier makes it simple to quickly find multicore performance bottlenecks and can help developers speed up the process of identifying and fixing such problems. Achieving perfect load balance is non-trivial and depends on the parallelism within the application, workload, and the threading implementation.

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

Generally speaking, mapping or scheduling of independent tasks (loads) to threads can happen in two ways: static and dynamic. When all tasks are the same length, a simple static division of tasks among available threads, dividing the total number of tasks into equal-sized groups assigned to each thread, is the best solution. Alternately, when the lengths of individual tasks differ, dynamic assignment of tasks to threads yields a better solution.

Concurrency analysis provided by Intel Parallel Amplifier measures how the application utilizes the available cores on a given system. During the analysis, Parallel Amplifier collects and provides information on how many threads are active, meaning threads which are either running or queued and are not waiting at a defined waiting or blocking API. The number of running threads corresponds to the concurrency level of an application. By comparing the concurrency level with the number of processors, Intel Parallel Amplifier classifies how the application utilizes the processors in the system.

To demonstrate how Intel Parallel Amplifier can identify hotspots and load imbalance issues, a sample program written in C is used here. This program computes the potential energy of a system of particles based on the distance in three dimensions. This is a multithreaded application that uses native Win32* threads and creates the number of threads specified by the NUM_THREADS variable. The scope of this discussion does not include the introduction of Win32 threads, threading methodologies, or how to introduce threads. Rather, it demonstrates how Intel Parallel Amplifier can significantly help identify load imbalance and help in the development of scalable parallel applications.

```c
for (i = 0; i < NUM_THREADS; i++)
{
    bounds[0][i] = i * (NPARTS / NUM_THREADS);
    bounds[1][i] = (i + 1) * (NPARTS / NUM_THREADS);
}
for (j = 0; j < NUM_THREADS; j++)
{
    tNum[j] = j;
```
tHandle[j] = CreateThread(NULL, 0, tPoolComputePot, &tNum[j], 0, NULL);
}

DWORD WINAPI tPoolComputePot (LPVOID pArg) {
    int tid = *(int *)pArg;
    while (!done)
    {
        WaitForSingleObject (bSignal[tid], INFINITE);
        computePot (tid);
        SetEvent (eSignal[tid]);
    }
    return 0;
}

The routine, which each thread executes in parallel, is given below. In the computePot routine, each thread uses the stored boundaries indexed by the thread’s assigned identification number (tid) to fix the start and end range of particles to be used. After each thread initializes its iteration space (start and end values), it starts computing the potential energy of the particles:

void computePot (int tid) {
    int i, j, start, end;
    double lPot = 0.0;
    double distx, disty, distz, dist;
    start = bounds[0][tid];
    end = bounds[1][tid];

    for (i = start; i < end; i++)
    {
        for (j = 0; j < i-1; j++)
        {
            distx = pow ((r[0][j] - r[0][i]), 2);
            disty = pow ((r[1][j] - r[1][i]), 2);
            distz = pow ((r[2][j] - r[2][i]), 2);
            dist = sqrt (distx + disty + distz);
            lPot += 1.0 / dist;
        }
    }
    gPot[tid] = lPot;
}

Hotspot analysis is used to find the hotspot(s) in the application so that the efforts can be focused on the appropriate function(s). The total elapsed time for this application is around 15.4 seconds on a single-socket system based on the Intel® Core™ 2 Quad processor. The hotspot analysis reveals that the computePot routine is the main hotspot, consuming most of the CPU time (23.331 seconds). Drilling down to the source of the computePot function shows the major contributors of this hotspot (Figure 1).
The concurrency analysis reveals that the CPU utilization on the same routine is poor (Figure 2) and the application uses 2.28 cores on average (Figure 3). The main hotspot is not utilizing all available cores; the CPU utilization is either poor (utilizing only one core) or OK (utilizing two to three cores) most of the time. The next question is whether there are any load imbalances that are contributing to the poor CPU utilization. The easiest way to find the answer is to select either Function-Thread-Bottom-up Tree or Thread-Function-Bottom-up Tree as the new granularity, as shown in Figure 4.

**Figure 1.** Source code view of the hotspot analysis.

**Figure 2.** Concurrency analysis results.
Figure 3. Summary view of the concurrency results.

The time values in the concurrency analysis results correspond to the following utilization types:

- **Idle**: All threads in the program are waiting; no threads are running.
- **Poor**: By default, poor utilization is defined as when the number of threads is up to 50 percent of the target concurrency.
- **OK**: By default, OK utilization is when the number of threads is between 51-85% of the target concurrency.
- **Ideal**: By default, ideal utilization is when the number of threads is between 86-115% of the target concurrency.

Figure 4. Concurrency analysis results showing Function->Thread grouping.

It is clear from Figure 4 that four worker threads executing this routine in parallel are not doing the same amount of work and thus contributing to the load imbalance and poor CPU utilization. Such behavior will prevent any multi-threaded application from scaling properly. A closer look at
the source code reveals that the outer loop within the main routine statically divides up the particle iterations based on the number of worker threads that will be created within the thread pool (start = bounds[0][tid], end = bound[1][tid]). The inner loop within this routine uses the outer index as the exit condition. Thus, the larger the particle number used in the outer loop, the more iterations of the inner loop will be executed. This is done so that each pair of particles contributes only once to the potential energy calculation. This static distribution clearly assigns different amounts of computation.

One way to fix this load imbalance issue is to use a more dynamic assignment of particles to threads. For example, rather than assigning consecutive groups of particles, as in the original version, each thread, starting with the particle indexed by the thread id (tid), can compute all particles whose particle number differs from the number of threads. For example, when using two threads, one thread handles the even-numbered particles, while the other thread handles the odd-numbered particles.

```c
void computePot(int tid) {
    int i, j;
    double lPot = 0.0;
    double distx, disty, distz, dist;
    for(i=tid; i<NPARTS; i+= NUM_THREADS ) { //<-for( i=start; i<end; i++)
        for( j=0; j<i-1; j++ ) {
            distx = pow( (r[0][j] - r[0][i]), 2 );
            disty = pow( (r[1][j] - r[1][i]), 2 );
            distz = pow( (r[2][j] - r[2][i]), 2 );
            dist = sqrt( distx + disty + distz );
            lPot += 1.0 / dist;
        }
    }
    gPot[tid] = lPot;
}
```

Analyzing the concurrency of the application after this change shows that the hotspot function is now fully utilizing all the cores available (Figure 5).

![Concurrency analysis results after the change.](image)

Figure 5. Concurrency analysis results after the change.

The summary pane (Figure 6) provides a quick review of the result and the effect of the change. Elapsed time dropped to ~9.0 seconds from ~15.4 seconds, and the average CPU utilization increased to 3.9 from 2.28. Simply enabling worker threads to perform equal amounts of computation enabled a 1.7x speedup, reducing the elapsed time by ~41.5%.
As can be seen from the summary and concurrency results, the new version of the application utilizes almost all of the available cores and both the serial code segments (poor utilization) and underutilization segments have disappeared.

**Advice**

There are various threading methods, and each method provides a different mechanism for handling the distribution of tasks to the threads. Some common threading methods include the following:

- Explicit or native threading methods (e.g., Win32 and POSIX* threads)
- Threading Abstraction
  - Intel® Threading Building Blocks
  - OpenMP*

Explicit threading methods (e.g., Win32 and POSIX 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, as shown in this example. For dynamic scheduling, two related methods are easily implemented: Producer/Consumer and Manager/Worker. In the former, one or more threads (Producer) place tasks into a queue, 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. In the Manager/Worker model, Worker threads rendezvous with the Manager thread, whenever more work is needed, to receive assignments directly.

Whatever 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. While a single Manager thread is easy to code and ensures proper distribution of tasks, should 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.

OpenMP provides four scheduling methods for iterative work-sharing constructs (see the OpenMP specification for a detailed description of each method). Static scheduling of iterations is used by default. When the amount of work per iteration varies and the pattern is unpredictable, dynamic scheduling of iterations can better balance the workload.

A microarchitectural issue called false sharing may arise when dynamic scheduling is used. False sharing is a performance-degrading pattern-access problem. False sharing happens when two or more threads repeatedly write to the same cache line (64 bytes on Intel architectures). Special care should be given when workloads are dynamically distributing among threads.

Intel® Threading Building Blocks (Intel® TBB) is a runtime-based parallel programming model, consisting of a template-based runtime library to help developers harness the latent performance of multicore processors. Intel TBB allows developers to write scalable applications that take advantage of concurrent collections and parallel algorithms. It provides a divide-and-conquer scheduling algorithm with a work-stealing mechanism so that the developers do not have to worry about various scheduling algorithms. By leveraging the work-stealing mechanism, Intel TBB balances tasks among worker threads dynamically.

Additional Resources
- Intel® Software Network Parallel Programming Community
- Intel® Parallel Studio
- OpenMP* Specifications
- Intel® Threading Building Blocks
- Intel Threading Building Blocks for Open Source