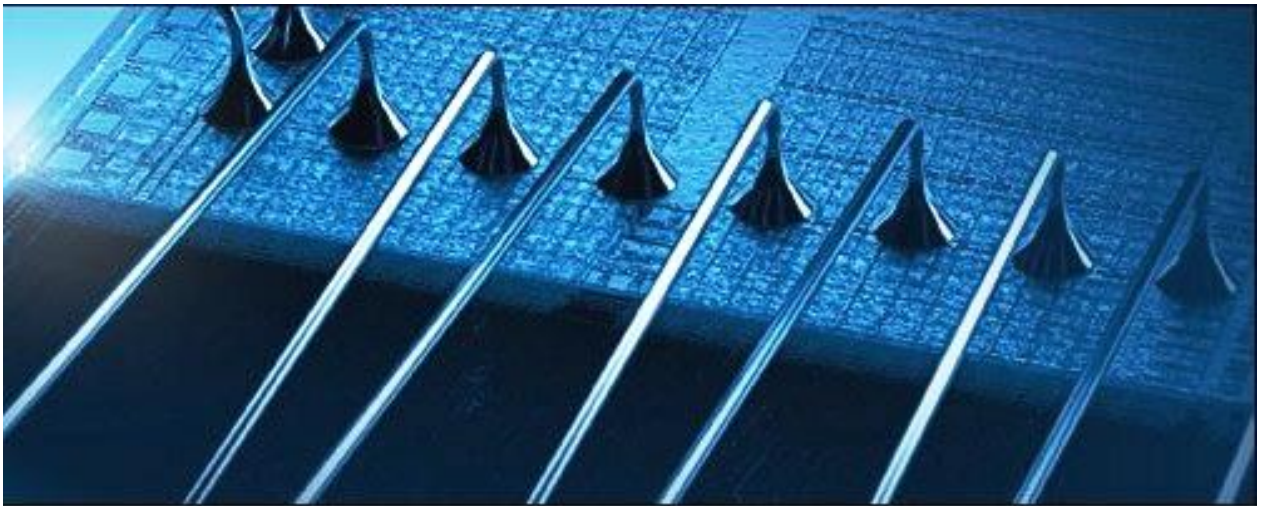


Intel Guide for Developing Multithreaded Application



<http://www.intel.com/software/threading-guide>

Disclaimers

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

UNLESS OTHERWISE AGREED IN WRITING BY INTEL, THE INTEL PRODUCTS ARE NOT DESIGNED NOR INTENDED FOR ANY APPLICATION IN WHICH THE FAILURE OF THE INTEL PRODUCT COULD CREATE A SITUATION WHERE PERSONAL INJURY OR DEATH MAY OCCUR.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or go to: <http://www.intel.com/design/literature.htm>

Copyright © 2011 Intel Corporation. All rights reserved

BunnyPeople, Celeron, Celeron Inside, Centrino, Centrino Inside, Core Inside, i960, Intel, the Intel logo, Intel AppUp, Intel Atom, Intel Atom Inside, Intel Core, Intel Inside, the Intel Inside logo, Intel NetBurst, Intel NetMerge, Intel NetStructure, Intel SingleDriver, Intel SpeedStep, Intel Sponsors of Tomorrow, the Intel Sponsors of Tomorrow. logo, Intel StrataFlash, Intel Viiv, Intel vPro, Intel XScale, InTru, the InTru logo, InTru soundmark, Itanium, Itanium Inside, MCS, MMX, Moblin, Pentium, Pentium Inside, skool, the skool logo, Sound Mark, The Journey Inside, vPro Inside, VTune, Xeon, and Xeon Inside are trademarks of Intel Corporation in the U.S. and other countries.

*Other names and brands may be claimed as the property of others.

The Authors

The Intel Guide for Developing Multithreaded Applications was written and edited by the following Intel Software Engineering Team members:

- Levent Akyil
- Jay Hoeflinger
- Paul Petersen
- Clay Breshears
- Richard Hubbard
- Todd Rosenquist
- Martyn Corden
- Alexey Kukanov
- Aaron Tersteeg
- Julia Fedorova
- Kevin O'Leary
- Vladimir Tsymbal
- Paul Fischer
- David Ott
- Mike Voss
- Henry Gabb
- Eric Palmer
- Thomas Zipplies
- Victoria Gromova
- Anton Pegushin

Visit the website for up-to-date articles:
www.intel.com/software/threading-guide

Optimization Notice

Intel® compilers, associated libraries and associated development tools may include or utilize options that optimize for instruction sets that are available in both Intel® and non-Intel microprocessors (for example SIMD instruction sets), but do not optimize equally for non-Intel microprocessors. In addition, certain compiler options for Intel compilers, including some that are not specific to Intel micro-architecture, are reserved for Intel microprocessors. For a detailed description of Intel compiler options, including the instruction sets and specific microprocessors they implicate, please refer to the “Intel® Compiler User and Reference Guides” under “Compiler Options.” Many library routines that are part of Intel® compiler products are more highly optimized for Intel microprocessors than for other microprocessors. While the compilers and libraries in Intel® compiler products offer optimizations for both Intel and Intel-compatible microprocessors, depending on the options you select, your code and other factors, you likely will get extra performance on Intel microprocessors.

Intel® compilers associated libraries and associated development tools may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include Intel® Streaming SIMD Extensions 2 (Intel® SSE2), Intel® Streaming SIMD Extensions 3 (Intel® SSE3), and Supplemental Streaming SIMD Extensions 3 (Intel® SSSE3) instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors.

While Intel believes our compilers and libraries are excellent choices to assist in obtaining the best performance on Intel® and non-Intel microprocessors, Intel recommends that you evaluate other compilers and libraries to determine which best meet your requirements. We hope to win your business by striving to offer the best performance of any compiler or library; please let us know if you find we do not.

Table of Contents

Introduction	3
Predicting and Measuring Parallel Performance.....	5
Loop Modifications to Enhance Data-Parallel Performance .	9
Granularity and Parallel Performance	14
Load Balance and Parallel Performance.....	20
Expose Parallelism by Avoiding or Removing Artificial Dependencies	24
Using Tasks Instead of Threads	27
Exploiting Data Parallelism in Ordered Data Streams	30
Using AVX without Writing AVX Code	35
Managing Lock Contention: Large and Small Critical Sections	41
Use Synchronization Routines Provided by the Threading API Rather than Hand-Coded Synchronization.....	45
Choosing Appropriate Synchronization Primitives to Minimize Overhead	48
Use Non-blocking Locks When Possible	52
Avoiding Heap Contention Among Threads.....	55
Use Thread-local Storage to Reduce Synchronization	60

Detecting Memory Bandwidth Saturation in Threaded Applications.....	64
Avoiding and Identifying False Sharing Among Threads	68
Optimizing Applications for NUMA	73
Automatic Parallelization with Intel® Compilers.....	78
Parallelism in the Intel® Math Kernel Library	82
Threading and Intel® Integrated Performance Primitives ...	86
Using Intel® Inspector XE 2011 to Find Data Races in Multithreaded Code.....	92
Curing Thread Imbalance Using Intel® Parallel Amplifier ...	98
Getting Code Ready for Parallel Execution with Intel® Parallel Composer.....	105
Optimize Data Structures and Memory Access Patterns to Improve Data Locality	121

Introduction

Motivation

The objective of the *Intel® Guide for Developing Multithreaded Applications* is to provide guidelines for developing efficient multithreaded applications across Intel-based symmetric multiprocessors (SMP) and/or systems with Hyper-Threading Technology. An application developer can use the advice in this document to improve multithreading performance and minimize unexpected performance variations on current as well as future SMP architectures built with Intel® processors.

The Guide provides general advice on multithreaded performance. Hardware-specific optimizations have deliberately been kept to a minimum. Topics covering hardware-specific optimizations may be added in the future for developers willing to sacrifice portability for higher performance.

Prerequisites

Readers should have programming experience in a high-level language, preferably C, C++, and/or Fortran, though many of the recommendations in this document also apply to languages such as 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.

Scope

The main objective of the Guide is to provide a quick reference to design and optimization guidelines for multithreaded applications on Intel® platforms. This Guide is not intended to serve as a textbook on multithreading nor is it a porting guide to Intel platforms.

Organization

The *Intel® Guide for Developing Multithreaded Applications* covers topics ranging from general advice applicable to any multithreading method to usage guidelines for Intel® software products to API-specific issues. Each topic in the *Intel® Guide for Developing Multithreaded Applications* is designed to stand on its own. However, the topics fall naturally into categories.

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 ($1 - \textit{pctPar}$), 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:

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

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 ($1 - pctPar$) 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 (\leq) 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:

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

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:

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

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.

Loop Modifications to Enhance Data-Parallel Performance

Abstract

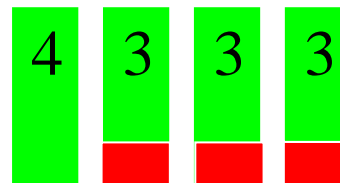
In data-parallel applications, the same independent operation is performed repeatedly on different data. Loops are usually the most compute-intensive segments of data parallel applications, so loop optimizations directly impact performance. When confronted with nested loops, the granularity of the computations 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. Still, the amount of work to be done per iteration should also be considered. Unless stated otherwise, the discussion in this section assumes that the amount of computation within each iteration of a loop is (roughly) equal to every other iteration in the same loop.

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

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



Advice

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

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

```

        for (int i = 0; i < imx; i++)
            ws[nv][k][j][i] = Process(w[nv][k][j][i]);
    }

```

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

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

```

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

```

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

If a loop has a loop-carried dependence that prevents the loop from being executed in parallel, it may be possible to break up the body of the loop into separate loops 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:

```

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

```

The assignment of elements within the `a` array are all independent, regardless of the sign of the corresponding elements of `b`. Each assignment of an element in `b` is independent of any other

assignment, but depends on the completion of the assignment of the required element of `a`. Thus, as written, the loop above cannot be parallelized.

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

```
float *a, *b;

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

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

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

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

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

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

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

For each iteration of the outermost loop in the above code, the full set of iterations of the `i`-loop will execute. From the selected element of the prime array, the start and end indices within the chunk of the `marked` array (controlled by the outer loop) must be found. These computations have been encapsulated within the `f()` and `g()` routines. Thus, the same chunk of `marked` will

be processed before the next one is processed. And, since the processing of each chunk is independent of any other, the iteration of the outer loop can be made to run in parallel.

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

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

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:

<pre>for (j = 0; j < N; j++) a[j] = b[j] + c[j]; for (j = 0; j < N; j++) d[j] = e[j] + f[j]; for (j = 5; j < N - 5; j++) g[j] = d[j+1] + a[j+1];</pre>	<pre>for (j = 0; j < N; j++) { a[j] = b[j] + c[j]; d[j] = e[j] + f[j]; } for (j = 5; j < N - 5; j++) g[j] = d[j+1] + a[j+1];</pre>
---	---

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

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

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

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

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

Additional Resources

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

[OpenMP* Specifications](#)

[Intel® Threading Building Blocks](#)

[Intel Threading Building Blocks for Open Source](#)

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

Granularity and Parallel Performance

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.

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.

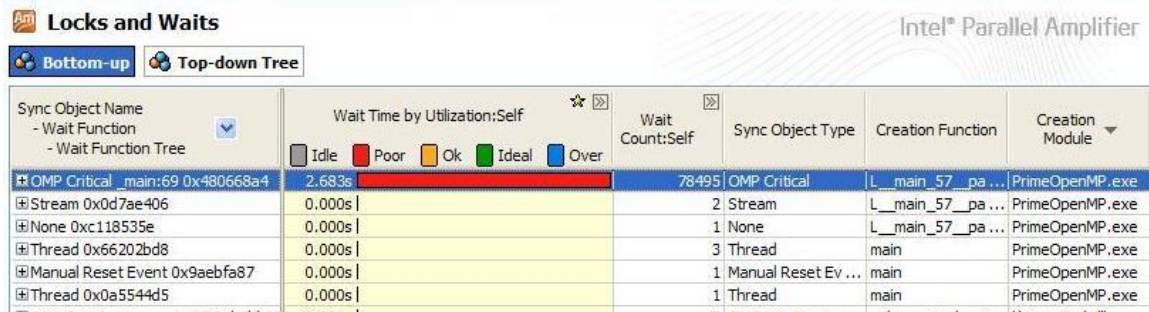


Figure 1. Locks and Waits analysis results demonstrating that the OpenMP* critical region is cause of synchronization overhead.

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.

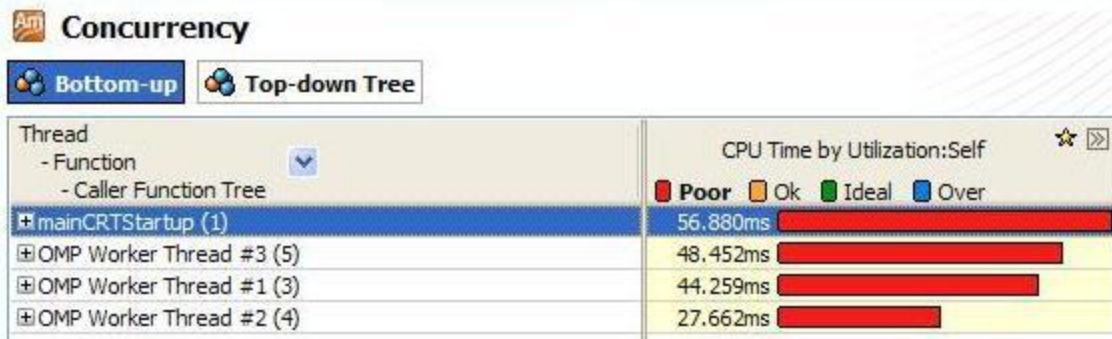


Figure 2. Concurrency analysis results demonstrating the imbalance of execution time used by each thread.

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.

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).

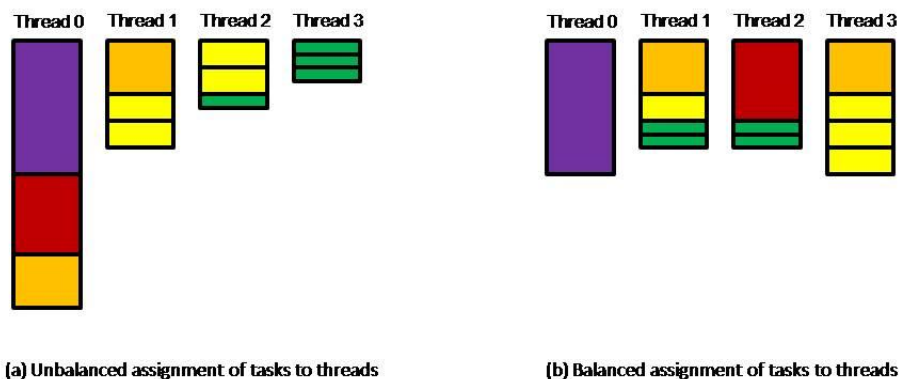


Figure 1. Examples of task distribution among four threads.

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, 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.

```
#pragma omp parallel for schedule(dynamic, 5)
for (i = 0; i < n; i++)
{
    unknown_amount_of_work(i);
}
```

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.

```
#pragma omp parallel for schedule(guided, 8)
for (i = 0; i < n; i++)
{
    uneven_amount_of_work(i);
}
```

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.

```
#pragma omp parallel for schedule(static, 4)
for (i = 0; i < n; i++)
{
    process_lower_triangular_row(i);
}
```

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.

Expose Parallelism by Avoiding or Removing Artificial Dependencies

Abstract

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

Background

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

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

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

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

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

declare lineCache[3]
lineCache[0] = 0
BlurLine (line 1 of imageIn, lineCache[1])
for each line i in (imageIn)
  BlurLine (line i+1 of imageIn, lineCache[i mod 3])
  lineSums = lineCache[0] + lineCache[1] + lineCache[2]
  lineOut = lineSums / 3
```

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

Another common example is pointer offsets inside a loop:

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

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

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

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

Advice

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

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

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

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

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

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

```
ptr = &someArray[0]
```

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

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

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

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

Usage Guidelines

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

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

Additional Resources

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

[OpenMP* Specifications](#)

Using Tasks Instead of Threads

Abstract

Tasks are a light-weight 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. Three 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 $n \geq 10$, 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.

```
int fibTBB(int n) {
    if( n < 10 ) {
        return fibSerial(n);
    } else {
        int x, y;
        tbb::task_group g;
        g.run([&]{x=fib(n-1);}); // spawn a task
        g.run([&]{y=fib(n-2);}); // spawn another task
        g.wait();                // wait for both tasks to complete
        return x+y;
    }
}
```

```

    }
}

```

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

Other libraries that support tasks include 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.

```

int fibOpenMP( int n ) {
    int i, j;
    if( n < 10 ) {
        return fibSerial(n);
    } else {
        // spawn a task
        #pragma omp task shared( i ), untied
        i = fib( n - 1 );
        // spawn another task
        #pragma omp task shared( j ), untied
        j = fib( n - 2 );
        // wait for both tasks to complete
        #pragma omp taskwait
        return i + j;
    }
}

```

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 also 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:

```
// serial loop
for (int i = 0; i < 10000; ++i)
    a[i] = f(i) + g(i);

// parallel loop
tbb::parallel_for( 0, 10000, [&](int i) { a[i] = f(i) + g(i); } );
```

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.](#)

[OpenMP* Specifications](#)

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:

```
inFile = OpenFile ()  
outFile == InitializeOutputFile ()
```

```

WriteHeader (outFile)
outputBuffer = AllocateBuffer (bufferSize)
while (frame = ReadNextFrame (inFile))
{
    EncodeFrame (frame, outputBuffer)
    if (outputBuffer size > bufferThreshold)
        FlushBuffer(outputBuffer, outFile)
}
FlushBuffer (outputBuffer, outFile)

```

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:

```

WriteHeader (outFile)
while (block = ReadNextBlock (inFile))
{
    while(frame = ReadNextFrame (block))
    {
        EncodeFrame (frame, outputBuffer)
        if (outputBuffer size > bufferThreshold)
            FlushBuffer (outputBuffer, outFile)
    }
    FlushBuffer (outputBuffer, outFile)
}

```

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:

```

WriteHeader (outFile)
while (block = ReadNextBlock (inFile))
{
    while (frame = ReadNextFrame (block))
    {
        EncodeFrame (frame, outputBuffer)
    }
    FlushBuffer (outputBuffer, outFile)
}

```

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:

```

while (block = ReadNextBlock (inFile))
{
    WriteHeader (outputBuffer)
    while (frame = ReadNextFrame (block))
    {
        EncodeFrame (frame, outputBuffer)
    }
    FlushBuffer (outputBuffer, outFile)
}

```

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

```

// Create a team of threads with private
// copies of outputBuffer, block, and frame
// and shared copies of inFile and outFile
while (AcquireLock,
        block = ReadNextBlock (inFile),
        ReleaseLock, block)
{
    WriteHeader (outputBuffer)
    while (frame = ReadNextFrame (block))
    {
        EncodeFrame (frame, outputBuffer)
    }
    FlushBuffer (outputBuffer, outFile)
}

```

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.

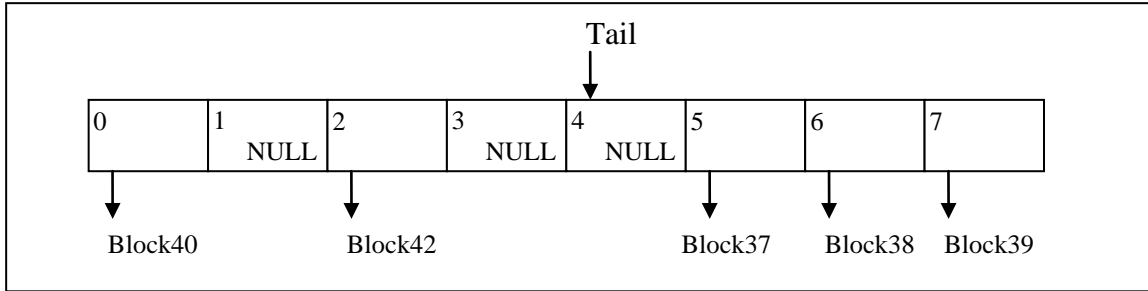


Figure 1. State of example reorder buffer before writing.

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.

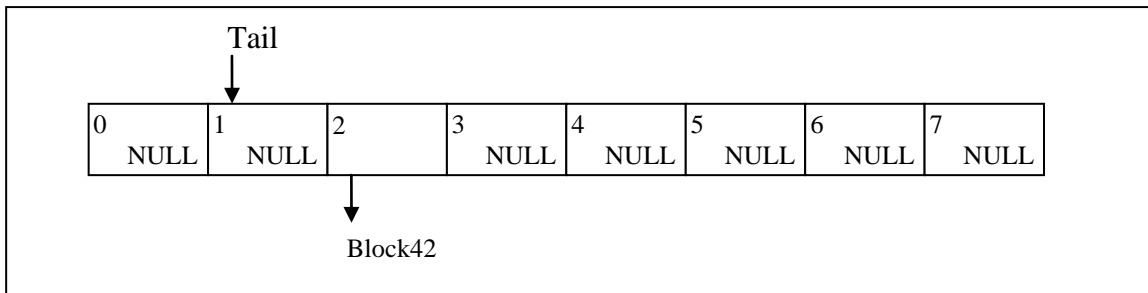


Figure 2. State of example reorder buffer after writing.

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:

```

inFile = OpenFile ()
outFile == InitializeOutputFile ()
// Create a team of threads with private
// copies of outputBuffer, block, and frame, shared
// copies of inFile and outFile.
while (AcquireLock,
       block = ReadNextBlock (inFile),
       ReleaseLock, block)
{
    WriteHeader (outputBuffer)
    while (frame = ReadNextFrame (block))

```

```

    {
        EncodeFrame (frame, outputBuffer)
    }
    QueueOrFlush (outputBuffer, outFile)
}

```

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](#)

Using AVX without Writing AVX Code

Abstract

Intel® Advanced Vector Extensions (Intel® AVX) is a new 256-bit instruction set extension to Intel® Streaming SIMD Extensions (Intel® SSE) and is designed for applications that are Floating Point (FP) intensive. Intel® SSE and Intel® AVX are both examples of Single Instruction Multiple Data instruction sets. Intel® AVX was released as part of the 2nd generation Intel® Core™ processor family. Intel® AVX improves performance due to wider 256-bit vectors, a new extensible instruction format (Vector Extension or VEX), and by its rich functionality.

The instruction set architecture supports three operands which improves instruction programming flexibility and allows for non-destructive source operands. Legacy 128-bit SIMD instructions have also been extended to support three operands and the new instruction encoding format (VEX). An instruction encoding format describes the way that higher-level instructions are expressed in a format the processor understands using opcodes and prefixes. This results in better management of data and general purpose applications like those for image, audio/video processing, scientific simulations, financial analytics and 3D modeling and analysis.

This article discusses options that developers can choose from to integrate Intel® AVX into their applications without explicitly coding in low-level assembly language. The most direct way that a C/C++ developer can access the features of Intel® AVX is to use the C-compatible intrinsic instructions. The intrinsic functions provide access to the Intel® AVX instruction set and to higher-level math functions in the Intel® Short Vector Math Library (SVML). These functions are declared in the `immintrin.h` and `ia32intrin.h` header files respectively. There are several other ways that an application programmer can utilize Intel® AVX without explicitly adding Intel® AVX instructions to their source code. This document presents a survey of these methods using the Intel® C++ Composer XE 2011 targeting execution on a Sandy Bridge system. The Intel® C++ Composer XE is supported on Linux*, Windows*, and Mac OS* X platforms. Command line switches for the Windows* platform will be used throughout the discussion.

Background

A vector or SIMD enabled-processor can simultaneously execute an operation on multiple data operands in a single instruction. An operation performed on a single number by another single number to produce a single result is considered a scalar process. An operation performed simultaneously on N numbers to produce N results is a vector process ($N > 1$). This technology is available on Intel processors or compatible, non-Intel processors that support SIMD or AVX instructions. The process of converting an algorithm from a scalar to vector implementation is called vectorization.

Advice

Recompile for Intel® AVX

The first method to consider is to simply recompile using the `/QaxAVX` compiler switch. The source code does not have to be modified at all. The Intel® Compiler will generate appropriate 128 and 256-bit Intel® AVX VEX-encoded instructions. The Intel® Compiler will generate multiple, processor-specific, auto-dispatched code paths for Intel processors when there is a performance benefit. The most appropriate code will be executed at run time.

Compiler Auto-vectorization

Compiling the application with the appropriate architecture switch is a great first step toward building Intel® AVX ready applications. The compiler can do much of the vectorization work on behalf of the software developer via auto-vectorization. Auto-vectorization is an optimization performed by compilers when certain conditions are met. The Intel® C++ Compiler can perform the appropriate vectorization automatically during code generation. An excellent document that describes vectorization in more detail can be found at [A Guide to Vectorization with Intel\(R\) C++ Compilers](#). The Intel Compiler will look for vectorization opportunities whenever the optimization level is /O2 or higher.

Let's consider a simple matrix-vector multiplication example that is provided with the Intel® C++ Composer XE that illustrates the concepts of vectorization. The following code snippet is from the matvec function in Multiply.c of the vec_samples archive:

```
void matvec(int size1, int size2, FTYPE a[][size2], FTYPE b[], FTYPE x[])
{
    for (i = 0; i < size1; i++) {
        b[i] = 0;

        for (j = 0; j < size2; j++) {
            b[i] += a[i][j] * x[j];
        }
    }
}
```

Without vectorization, the outer loop will execute size1 times and the inner loop will execute size1*size2 times. After vectorization with the /QaxAVX switch the inner loop can be unrolled because four multiplications and four additions can be performed in a single instruction per operation. The vectorized loops are much more efficient than the scalar loop. The advantage of Intel® AVX also applies to single-precision floating point numbers as eight single-precision floating point operands can be held in a ymm register.

Loops must meet certain criteria in order to be vectorized. The loop trip count has to be known when entering the loop at runtime. The trip count can be a variable, but it must be constant while executing the loop. Loops have to have a single entry and single exit, and exit cannot be dependent on input data. There are some branching criteria as well, e.g., switch statements are not allowed. If-statements are allowed provided that they can be implemented as masked assignments. Innermost loops are the most likely candidates for vectorization, and the use of function calls within the loop can limit vectorization. Inlined functions and intrinsic SVML functions increase vectorization opportunities.

It is recommended to review vectorization information during the implementation and tuning stages of application development. The Intel® Compiler provides vectorization reports that provide insight into what was and was not vectorized. The reports are enabled via /Qvec-report=<n> command line option, where n specifies the level of reporting detail. The level of detail increases with a larger value of n. A value of n=3 will provide dependency information, loops vectorized, and loops not vectorized. The developer can modify the implementation based on the information provided in the report; the reason why a loop was not vectorized is very helpful.

The developer's intimate knowledge of his or her specific application can sometimes be used to override the behavior of the auto-vectorizer. Pragmas are available that provide additional information to assist with the auto-vectorization process. Some examples are: to always vectorize a loop, to specify that the data within a loop is aligned, to ignore potential data dependencies, etc. The addFloats example illustrates some important points. It is necessary to

review the generated assembly language instructions to see what the compiler generated. The Intel Compiler will generate an assembly file in the current working directory when the /S command line option is specified.

```
void addFloats(float *a, float *b, float *c, float *d, float *e, int n) {
    int i;
    #pragma simd
    #pragma vector aligned
    for(i = 0; i < n; ++i) {
        a[i] = b[i] + c[i] + d[i] + e[i];
    }
}
```

Note the use of the `simd` and `vector` pragmas. They play a key role to achieve the desired Intel® AVX 256-bit vectorization. Adding “`#pragma simd`” to the code helps as packed versions of Intel® 128-bit AVX instructions are generated. The compiler also unrolled the loop once which reduces the number of executed instructions related to end-of-loop testing. Specifying “`pragma vector aligned`” provides another hint that instructs the compiler to use aligned data movement instructions for all array references. The desired 256-bit Intel® AVX instructions are generated by using both “`pragma simd`” and “`pragma vector aligned`.” The Intel® Compiler chose `vmovups` because there is no penalty for using the unaligned move instruction when accessing aligned memory on the 2nd generation Intel® Core™ processor.

With `#pragma simd` and `#pragma vector aligned`

```
.B46.4::
    vmovups    ymm0, YMMWORD PTR [rdx+rax*4]
    vaddps     ymm1, ymm0, YMMWORD PTR [r8+rax*4]
    vaddps     ymm2, ymm1, YMMWORD PTR [r9+rax*4]
    vaddps     ymm3, ymm2, YMMWORD PTR [rbx+rax*4]
    vmovups    YMMWORD PTR [rcx+rax*4], ymm3
    vmovups    ymm4, YMMWORD PTR [32+rdx+rax*4]
    vaddps     ymm5, ymm4, YMMWORD PTR [32+r8+rax*4]
    vaddps     ymm0, ymm5, YMMWORD PTR [32+r9+rax*4]
    vaddps     ymm1, ymm0, YMMWORD PTR [32+rbx+rax*4]
    vmovups    YMMWORD PTR [32+rcx+rax*4], ymm1
    add        rax, 16
    cmp        rax, r11
    jb         .B46.4
```

This demonstrates some of the auto-vectorization capabilities of the Intel® Compiler. Vectorization can be confirmed by vector reports, the `simd assert` pragma, or by inspection of generated assembly language instructions. Pragmas can further assist the compiler when used by developers with a thorough understanding of their applications. Please refer to [A Guide to Vectorization with Intel\(R\) C++ Compilers](#) for more details on vectorization in the Intel Compiler. The [Intel® C++ Compiler XE 12.0 User and Reference Guide](#) has additional information on the use of vectorization, pragmas and compiler switches. The Intel Compiler can do much of the vectorization work for you so that your application will be ready to utilize Intel® AVX.

Intel® Cilk™ Plus C/C++ Extensions for Array Notations

The Intel® Cilk™ Plus C/C++ language extension for array notations is an Intel-specific language extension that is applicable when an algorithm operates on arrays, and doesn't require a specific ordering of operations among the elements of the array. If the algorithm is expressed using array notation and compiled with the AVX switch, the Intel® Compiler will generate Intel® AVX instructions. C/C++ language extensions for array notations are intended to allow users to directly express high-level parallel vector array operations in their programs. This assists the compiler in performing data dependence analysis, vectorization, and auto-parallelization. From the developer's point of view, they will see more predictable vectorization, improved performance and better hardware resource utilization. The combination of C/C++ language extension for array notations and other Intel® Cilk™ Plus language extensions simplify parallel and vectorized application development.

The developer begins by writing a standard C/C++ elemental function that expresses an operation using scalar syntax. This elemental function can be used to operate on a single element when invoked without C/C++ language extension for array notation. The elemental function must be declared using "`__declspec(vector)`" so that it can also be invoked by callers using C/C++ language extension for array notation.

The `multiplyValues` example is shown as an elemental function:

```
__declspec(vector) float multiplyValues(float b, float c)
{
    return b*c;
}
```

This scalar invocation is illustrated in this simple example:

```
float a[12], b[12], c[12];
a[j] = multiplyValues(b[j], c[j]);
```

The function can also act upon an entire array, or portion of an array, by utilizing of C/C++ language extension for array notations. A section operator is used to describe the portion of the array on which to operate. The syntax is: `[<lower bound> : <length> : <stride>]`

Where lower bound is the starting index of the source array, length is the length of the resultant array, and stride expresses the stride through the source array. Stride is optional and defaults to one.

These array section examples will help illustrate the use:

```
float a[12];
```

`a[:]` refers to the entire a array

`a[0:2:3]` refers to elements 0 and 3 of array a.

`a[2:2:3]` refers to elements 2 and 5 of array a

`a[2:3:3]` refers to elements 2, 5, and 8 of array a

The notation also supports multi-dimensional arrays.

```
float a2d[12][4];
```

`a2d[:][:]` refers to the entire a2d array

`a2d[:][0:2:2]` refers to elements 0 and 2 of the columns for all rows of a2d.

The array notation makes it very straightforward to invoke the `multiplyValues` using arrays. The Intel® Compiler provides the vectorized version and dispatches execution appropriately. Here are some examples. The first example acts on the entire array and the second operates on a subset or section of the array.

This example invokes the function for the entire array:

```
a[:] = multiplyValues(b[:], c[:]);
```

This example invokes the function for a subset of the arrays:

```
a[0:5] = multiplyValues(b[0:5], c[0:5]);
```

These simple examples show how C/C++ language extension for array notations uses the features of Intel® AVX without requiring the developer to explicitly use any Intel® AVX instructions. C/C++ language extension for array notations can be used with or without elemental functions. This technology provides more flexibility and choices to developers, and utilizes the latest Intel® AVX instruction set architecture. Please refer to the [Intel® C++ Compiler XE 12.0 User and Reference Guide](#) for more details on Intel® Cilk™ Plus C/C++ language extension for array notations.

Using the Intel® IPP and Intel® MKL Libraries

Intel offers thousands of highly optimized software functions for multimedia, data processing, cryptography, and communications applications via the Intel® Integrated Performance Primitives and Intel® Math Kernel Libraries. These thread-safe libraries support multiple operating systems and the fastest code will be executed on a given platform. This is an easy way to add multi-core parallelization and vectorization to an application, as well as take advantage of the latest instructions available for the processor executing the code. The Intel® Performance Primitives 7.0 includes approximately 175 functions that have been optimized for Intel® AVX. These functions can be used to perform FFT, filtering, convolution, correlation, resizing and other operations. The Intel® Math Kernel Library 10.2 introduced support for Intel® AVX for BLAS (DGEMM), FFT, and VML (exp, log, pow). The implementation has been simplified in Intel® MKL 10.3 as the initial call to `mkl_enable_instructions` is no longer necessary. Intel® MKL 10.3 extended Intel® AVX support to DGMM/SGEMM, radix-2 Complex FFT, most real VML functions, and VSL distribution generators.

If you are already using, or are considering using these versions of the libraries, then your application will be able to utilize the Intel® AVX instruction set. The libraries will execute Intel® AVX instructions when run on a Sandy Bridge platform and are supported on Linux*, Windows*, and Mac OS* X platforms.

More information on the Intel® IPP functions that have been optimized for Intel® AVX can be found in <http://software.intel.com/en-us/articles/intel-ipp-functions-optimized-for-intel-avx-intel-advanced-vector-extensions/>. More information on Intel® MKL AVX support can be found in [Intel\(R\) AVX Optimization in Intel\(R\) MKL V10.3](#).

Usage Guidelines

The need for greater computing performance continues to drive Intel's innovation in micro-architectures and instruction sets. Application developers want to ensure that their product will be able to take advantage of advancements without a significant development effort. The methods, tools, and libraries discussed in this paper provide the means for developers to benefit from the advancements introduced by Intel® Advanced Vector Extensions without having to write a single line of Intel® AVX assembly language.

Additional Resources

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

[Intel\(R\) Advanced Vector Extensions](#)

[Using AVX Without Writing AVX](#)

[Intel\(R\) Compilers](#)

[Intel\(R\) C++ Composer XE 2011 - Documentation](#)

[How to Compile for Intel\(R\) AVX](#)

[A Guide to Vectorization with Intel\(R\) C++ Compilers](#)

[Intel\(R\) Integrated Performance Primitives Functions Optimized for Intel\(R\) Advanced Vector Extensions](#)

[Enabling Intel\(R\) Advanced Vector Extensions Optimizations in Intel\(R\) MKL](#)

[Intel\(R\) AVX Optimization in Intel\(R\) MKL V10.3](#)

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.

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:

```
Begin Thread Function ()
    Initialize ()

    BEGIN CRITICAL SECTION 1
        UpdateSharedData1 ()
    END CRITICAL SECTION 1

    DoFunc1 ()

    BEGIN CRITICAL SECTION 2
        UpdateSharedData2 ()
    END CRITICAL SECTION 2

    DoFunc2 ()
End Thread Function ()
```

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:

```
Begin Thread Function ()
    Initialize ()

    BEGIN CRITICAL SECTION 1
```

```

        UpdateSharedData1 ()
        DoFunc1 ()
        UpdateSharedData2 ()
    END CRITICAL SECTION 1

    DoFunc2 ()
End Thread Function ()

```

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:

```

Begin Thread Function ()
    Initialize ()

    BEGIN CRITICAL SECTION 1
        UpdateSharedData1 ()
    END CRITICAL SECTION 1

    DoFunc1 ()

    BEGIN CRITICAL SECTION 2
        UpdateSharedData2 ()
    END CRITICAL SECTION 2

```



```

BEGIN CRITICAL SECTION 3
    UpdateSharedData3 ()
END CRITICAL SECTION 3

DoFunc2 ()
End Thread Function ()

```

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:

```

int gNumThreads;
CRITICAL_SECTION gCs;

int main ()
{
    int spinCount = 0;
    ...
    spinCount = gNumThreads * 1000;
    if (spinCount > 8000) spinCount = 8000;
    InitializeCriticalSectionAndSpinCount (&gCs, spinCount);
    ...
}

DWORD WINAPI ThreadFunc (void *data)

```

```
{  
    ...  
    EnterCriticalSection (&gCs);  
    ...  
    LeaveCriticalSection (&gCs);  
}
```

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](#)

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.

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:

```
#include <ia64intrin.h>

void acquire_lock (int *lock)
{
    while (_InterlockedCompareExchange (lock, TRUE, FALSE) == TRUE);
}

void release_lock (int *lock)
{
    *lock = FALSE;
}
```

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 a `queuing_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.

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.

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:

```
#include <windows.h>

CRITICAL_SECTION cs; /* InitializeCriticalSection called in main()
*/
HANDLE mtx; /* CreateMutex called in main() */
static LONG counter = 0;

void IncrementCounter ()
{
    // Synchronize with Win32 interlocked function
    InterlockedIncrement (&counter);

    // Synchronize with Win32 critical section
    EnterCriticalSection (&cs);
    counter++;
    LeaveCriticalSection (&cs);

    // Synchronize with Win32 mutex
    WaitForSingleObject (mtx, INFINITE);
    counter++;
    ReleaseMutex (mtx);
}
```

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`:

```
Node* FreeList;
typedef spin_mutex FreeListMutexType;
FreeListMutexType FreeListMutex;

Node* AllocateNode()
{
    Node* n;
    {
        FreeListMutexType::scoped_lock lock(FreeListMutex);
        n = FreeList;
        if( n )
            FreeList = n->next;
    }
    if( !n )
        n = new Node();
    return n;
}
```

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:

```
static LONG N = 0;
LONG localVar;
...
InterlockedIncrement (&N);
InterlockedIncrement (&N);
InterlockedExchange (&localVar, N);

static LONG N = 0;
LONG localVar;
...
EnterCriticalSection (&lock);
    localVar = (N += 2);
LeaveCriticalSection (&lock);
```

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 `HANDLES`, 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.

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.

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:

```
CRITICAL_SECTION cs;  
void threadfoo()  
{  
    while(TryEnterCriticalSection(&cs) == FALSE)  
    {
```

```

    // some useful work
}
    // Critical Section of Code
    LeaveCriticalSection (&cs);
}
    // other work
}

```

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.

```

void threadfoo ()
{
    DWORD ret_value;
    HANDLE hHandle;
    // Some work
    ret_value = WaitForSingleObject (hHandle,0);

    if (ret_value == WAIT_TIME_OUT)
    {
        // Thread could not gain ownership of the kernel
        // object within the time interval;
        // Some useful work
    }
    else if (ret_value == WAIT_OBJECT_0)
    {
        // Critical Section of Code
    }
    else { // Handle Wait Failure}
        // Some work
    }
}

```

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 see if the request has been successful before releasing the shared object.

Usage Guidelines

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

Win32 Multithreaded Programming, Aaron Cohen and Mike Woodring.

Multithreading Applications in Win32 – the Complete Guide to Threads, Jim Beveridge and Robert Wiener.

Multithreaded Programming with Pthreads, Bil Lewis and Daniel J Berg.

Avoiding Heap Contention Among Threads

Abstract

Allocating memory from the system heap can be an expensive operation due to a lock used by system runtime libraries to synchronize access to the heap. Contention on this lock can limit the performance benefits from multithreading. To solve this problem, apply an allocation strategy that avoids using shared locks, or use third party heap managers.

Background

The system heap (as used by `malloc`) is a shared resource. To make it safe to use by multiple threads, it is necessary to add synchronization to gate access to the shared heap. Synchronization (in this case lock acquisition), requires two interactions (i.e., locking and unlocking) with the operating system, which is an expensive overhead. Serialization of all memory allocations is an even bigger problem, as threads spend a great deal of time waiting on the lock, rather than doing useful work.

The screenshots from Intel® Parallel Amplifier in Figures 1 and 2 illustrate the heap contention problem in a multithreaded CAD application.

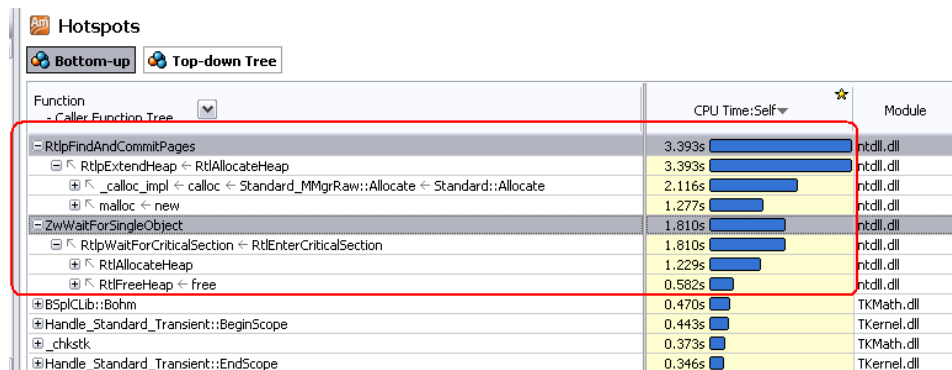


Figure 1. Heap allocation routines and kernel functions called from them are the bottleneck consuming most of the application execution time.

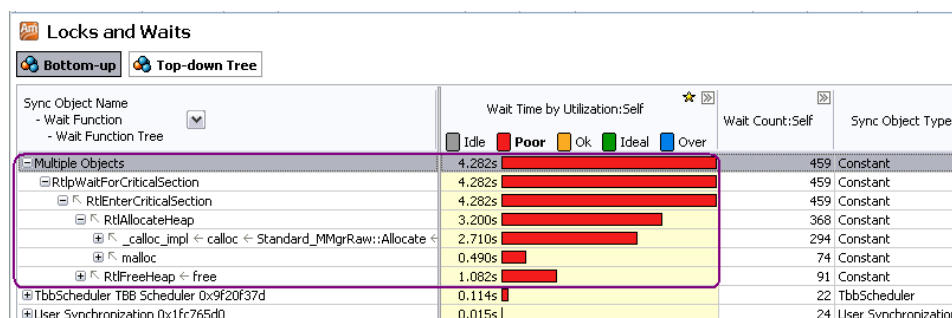


Figure 2. The critical section used in the heap allocation routines was the most contended synchronization object, causing a significant amount of wait time and poor CPU utilization.

Advice

The OpenMP* implementation in the Intel® Compilers exports two functions: `kmp_malloc` and `kmp_free`. These functions maintain a per-thread heap attached to each thread utilized by OpenMP and avoid the use of the lock that protects access to the standard system heap.

The Win32* API function `HeapCreate` can be used to allocate separate heaps for all of the threads used by the application. The flag `HEAP_NO_SERIALIZE` is used to disable the use of synchronization on this new heap, since only a single thread will access it. The heap handle can be stored in a Thread Local Storage (TLS) location in order to use this heap whenever an application thread needs to allocate or free memory. Note that memory allocated in this manner must be explicitly released by the same thread that performs the allocation.

The example below illustrates how to use the Win32 API features mentioned above to avoid heap contention. It uses a dynamic load library (.DLL) to register new threads at the point of creation, requests independently managed unsynchronized heap for each thread, and uses TLS to remember the heap assigned to the thread.

```
#include <windows.h>

static DWORD tls_key;

__declspec(dllexport) void *
thr_malloc( size_t n )
{
    return HeapAlloc( TlsGetValue( tls_key ), 0, n );
}

__declspec(dllexport) void
thr_free( void *ptr )
{
    HeapFree( TlsGetValue( tls_key ), 0, ptr );
}

// This example uses several features of the WIN32 programming API
// It uses a .DLL module to allow the creation and destruction of
// threads to be recorded.

BOOL WINAPI DllMain(
    HINSTANCE hinstDLL, // handle to DLL module
    DWORD fdwReason, // reason for calling function
    LPVOID lpReserved ) // reserved
{
    switch( fdwReason ) {
        case DLL_PROCESS_ATTACH:
            // Use Thread Local Storage to remember the heap
            tls_key = TlsAlloc();
            TlsSetValue( tls_key, GetProcessHeap() );
            break;

        case DLL_THREAD_ATTACH:
            // Use HEAP_NO_SERIALIZE to avoid lock overhead
            TlsSetValue( tls_key, HeapCreate( HEAP_NO_SERIALIZE, 0, 0 ) );
            break;
    }
}
```

```

case DLL_THREAD_DETACH:
    HeapDestroy( TlsGetValue( tls_key ) );
    break;

case DLL_PROCESS_DETACH:
    TlsFree( tls_key );
    break;
}
return TRUE; // Successful DLL_PROCESS_ATTACH.
}

```

The `pthread_key_create` and `pthread_{get|set}specific` API can be used to obtain access to TLS in applications using POSIX* threads (Pthreads*), but there is no common API to create independent heaps. It is possible to allocate a big portion of memory for each thread and store its address in TLS, but the management of this storage is the programmer's responsibility.

In addition to the use of multiple independent heaps, it is also possible to incorporate other techniques to minimize the lock contention caused by a shared lock that is used to protect the system heap. If the memory is only to be accessed within a small lexical context, the `alloca` routine can sometimes be used to allocate memory from the current stack frame. This memory is automatically deallocated upon function return.

```

// Uses of malloc() can sometimes be replaced with alloca()
{
    ...
    char *p = malloc( 256 );

    // Use the allocated memory
    process( p );

    free( p );
    ...
}

// If the memory is allocated and freed in the same routine.
{
    ...
    char *p = alloca( 256 );

    // Use the allocated memory
    process( p );
    ...
}

```

Note, however that Microsoft has deprecated `_alloca` and recommends using the security enhanced routine called `_malloca` instead. It allocates either from the stack or from the heap depending on the requested size; therefore, the memory obtained from `_malloca` should be released with `_freea`.

A per-thread free list is another technique. Initially, memory is allocated from the system heap with `malloc`. When the memory would normally be released, it is added to a per-thread linked list. If the thread needs to reallocate memory of the same size, it can immediately retrieve the stored allocation from the list without going back to the system heap.

```

struct MyObject {
    struct MyObject *next;
    ...
};

// the per-thread list of free memory objects
static __declspec(thread)
struct MyObject *freelist_MyObject = 0;

struct MyObject *
malloc_MyObject( )
{
    struct MyObject *p = freelist_MyObject;

    if (p == 0)
        return malloc( sizeof( struct MyObject ) );

    freelist_MyObject = p->next;

    return p;
}

void
free_MyObject( struct MyObject *p )
{
    p->next = freelist_MyObject;
    freelist_MyObject = p;
}

```

If the described techniques are not applicable (e.g., the thread that allocates the memory is not necessarily the thread that releases the memory) or memory management still remains a bottleneck, then it may be more appropriate to look into using a third party replacement to the heap manager. Intel® Threading Building Blocks (Intel® TBB) offers a multithreading-friendly memory manager that can be used with Intel TBB-enabled applications as well as with OpenMP and manually threaded applications. Some other third-party heap managers are listed in the Additional Resources section at the end of this article.

Usage Guidelines

With any optimization, you encounter trade-offs. In this case, the trade-off is in exchanging lower contention on the system heap for higher memory usage. When each thread is maintaining its own private heap or collection of objects, these areas are not available to other threads. This may result in a memory imbalance between the threads, similar to the load imbalance you encounter when threads are performing varying amounts of work. The memory imbalance may cause the working set size and the total memory usage by the application to increase. The increase in memory usage usually has a minimal performance impact. An exception occurs when the increase in memory usage exhausts the available memory. If this happens, it may cause the application to either abort or swap to disk.

Additional Resources

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

[Microsoft Developer Network: HeapAlloc, HeapCreate, HeapFree](#)

[Microsoft Developer Network: TlsAlloc, TlsGetValue, TlsSetValue](#)

[Microsoft Developer Network: alloca, malloc, free](#)

[MicroQuill SmartHeap for SMP](#)

[The HOARD memory allocator](#)

[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.

Use Thread-local Storage to Reduce Synchronization

Abstract

Synchronization is often an expensive operation that can limit the performance of a multi-threaded program. Using thread-local data structures instead of data structures shared by the threads can reduce synchronization in certain cases, allowing a program to run faster.

Background

When data structures are shared by a group of threads and at least one thread is writing into them, synchronization between the threads is sometimes necessary to make sure that all threads see a consistent view of the shared data. A typical synchronized access regime for threads in this situation is for a thread to acquire a lock, read or write the shared data structures, then release the lock.

All forms of locking have overhead to maintain the lock data structures, and they use atomic instructions that slow down modern processors. Synchronization also slows down the program, because it eliminates parallel execution inside the synchronized code, forming a serial execution bottleneck. Therefore, when synchronization occurs within a time-critical section of code, code performance can suffer.

The synchronization can be eliminated from the multithreaded, time-critical code sections if the program can be re-written to use thread-local storage instead of shared data structures. This is possible if the nature of the code is such that real-time ordering of the accesses to the shared data is unimportant. Synchronization can also be eliminated when the ordering of accesses is important, if the ordering can be safely postponed to execute during infrequent, non-time-critical sections of code.

Consider, for example, the use of a variable to count events that happen on several threads. One way to write such a program in OpenMP* is as follows:

```
int count=0;
#pragma omp parallel shared(count)
{
    . . .
    if (event_happened) {
#pragma omp atomic
        count++;
    }
    . . .
}
```

This program pays a price each time the event happens, because it must synchronize to guarantee that only one thread increments `count` at a time. Every event causes synchronization. Removing the synchronization makes the program run faster. One way to do this is to have each thread count its own events during the parallel region and then sum the individual counts later. This technique is demonstrated in the following program:

```
int count=0;
int tcount=0;
#pragma omp threadprivate(tcount)

omp_set_dynamic(0);
```

```

#pragma omp parallel
{
    . . .
    if (event_happened) {
        tcount++;
    }
    . . .
}
#pragma omp parallel shared(count)
{
    #pragma omp atomic
    count += tcount;
}

```

This program uses a `tcount` variable that is private to each thread to store the count for each thread. After the first parallel region counts all the local events, a subsequent region adds this count into the overall count. This solution trades synchronization per event for synchronization per thread. Performance will improve if the number of events is much larger than the number of threads. Please note that this program assumes that both parallel regions execute with the same number of threads. The call to `omp_set_dynamic(0)` prevents the number of threads from being different than the number requested by the program.

An additional advantage of using thread-local storage during time-critical portions of the program is that the data may stay live in a processor's cache longer than shared data, if the processors do not share a data cache. When the same address exists in the data cache of several processors and is written by one of them, it must be invalidated in the caches of all other processors, causing it to be re-fetched from memory when the other processors access it. But thread-local data will never be written by any other processors than the one it is local to and will therefore be more likely to remain in the cache of its processor.

The code snippet above shows one way to specify thread-local data in OpenMP. To do the same thing with Pthreads, the programmer must create a key for thread-local data, then access the data via that key. For example:

```

#include <pthread.h>

pthread_key_t tsd_key;
<arbitrary data type> value;

if( pthread_key_create(&tsd_key, NULL) ) err_abort(status, "Error
creating key");
if( pthread_setspecific( tsd_key, value))
    err_abort(status, "Error in pthread_setspecific");
. . .
value = (<arbitrary data type>)pthread_getspecific( tsd_key );

```

With Windows threads, the operation is very similar. The programmer allocates a TLS index with `TlsAlloc`, then uses that index to set a thread-local value. For example:

```

DWORD tls_index;

```

```

LPVOID value;

tls_index = TlsAlloc();
if (tls_index == TLS_OUT_OF_INDEXES) err_abort( tls_index, "Error in
TlsAlloc");
status = TlsSetValue( tls_index, value );
if ( status == 0 ) err_abort( status, "Error in TlsSetValue");
. . .
value = TlsGetValue( tls_index );

```

In OpenMP, one can also create thread-local variables by specifying them in a `private` clause on the `parallel` pragma. These variables are automatically deallocated at the end of the parallel region. Of course, another way to specify thread-local data, regardless of the threading model, is to use variables allocated on the stack in a given scope. Such variables are deallocated at the end of the scope.

Advice

The technique of thread-local storage is applicable if synchronization is coded within a time-critical section of code, and if the operations being synchronized need not be ordered in real-time. If the real-time order of the operations is important, then the technique can still be applied if enough information can be captured during the time-critical section to reproduce the ordering later, during a non-time-critical section of code.

Consider, for example, the following example, where threads write data into a shared buffer:

```

int buffer[NENTRIES];

main() {
    . . .

    #pragma omp parallel
    {
        . . .
        update_log(time, value1, value2);
        . . .
    }

    . . .
}

void update_log(time, value1, value2)
{
    #pragma omp critical
    {
        if (current_ptr + 3 > NENTRIES)
        { print_buffer_overflow_message(); }

        buffer[current_ptr] = time;
        buffer[current_ptr+1] = value1;
        buffer[current_ptr+2] = value2;
        current_ptr += 3;
    }
}

```

Assume that `time` is some monotonically increasing value and the only real requirement of the program for this buffer data is that it be written to a file occasionally, sorted according to time. Synchronization can be eliminated in the `update_log` routine by using thread-local buffers. Each thread would allocate a separate copy of `tpbuffer` and `tpcurrent_ptr`. This allows the elimination of the critical section in `update_log`. The entries from the various thread-private buffers can be merged later, according to the time values, in a non-time-critical portion of the program.

Usage Guidelines

One must be careful about the trade-offs involved in this technique. The technique does not remove the need for synchronization, but only moves the synchronization from a time-critical section of the code to a non-time-critical section of the code.

- First, determine whether the original section of code containing the synchronization is actually being slowed down significantly by the synchronization. Intel® Parallel Amplifier and/or Intel® VTune™ Performance Analyzer can be used to check each section of code for performance problems.
- Second, determine whether the time ordering of the operations is critical to the application. If not, synchronization can be removed, as in the event-counting code. If time ordering is critical, can the ordering be correctly reconstructed later?
- Third, verify that moving synchronization to another place in the code will not cause similar performance problems in the new location. One way to do this is to show that the number of synchronizations will decrease dramatically because of your work (such as in the event-counting example above).

Additional Resources

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

[OpenMP* Specifications](#)

David R. Butenhof, *Programming with POSIX Threads*, Addison-Wesley, 1997.

Johnson M. Hart, *Win32 System Programming* (2nd Edition), Addison-Wesley, 2001.

Jim Beveridge and Robert Weiner, *Multithreading Applications in Win32*, Addison-Wesley, 1997.

Detecting Memory Bandwidth Saturation in Threaded Applications

Abstract

Memory sub-system components contribute significantly to the performance characteristics of an application. As an increasing number of threads or processes share the limited resources of cache capacity and memory bandwidth, the scalability of a threaded application can become constrained. Memory-intensive threaded applications can suffer from memory bandwidth saturation as more threads are introduced. In such cases, the threaded application won't scale as expected, and performance can be reduced. This article introduces techniques to detect memory bandwidth saturation in threaded applications.

Background

As modern processors include more cores and bigger caches, they become faster at a higher rate than the memory sub-system components. The increasing core count on a per-die basis has put pressure on the cache capacity and memory bandwidth. As a result, optimally utilizing the available cache and memory bandwidth to each core is essential in developing forward-scaling applications. If a system isn't capable of moving data from main memory to the cores fast enough, the cores will sit idle as they wait for the data to arrive. An idle core during computation is a wasted resource that increases the overall execution time of the computation and will negate some of the benefits of more cores.

The current generation of Intel® processors based on the Nehalem architecture moved from the traditional front-side-bus (FSB) approach to non-uniform memory access/architecture (NUMA) model to increase the available memory bandwidth to the cores and reduce the bandwidth saturation issues mentioned above. Figure 1 depicts the FSB to NUMA transition.

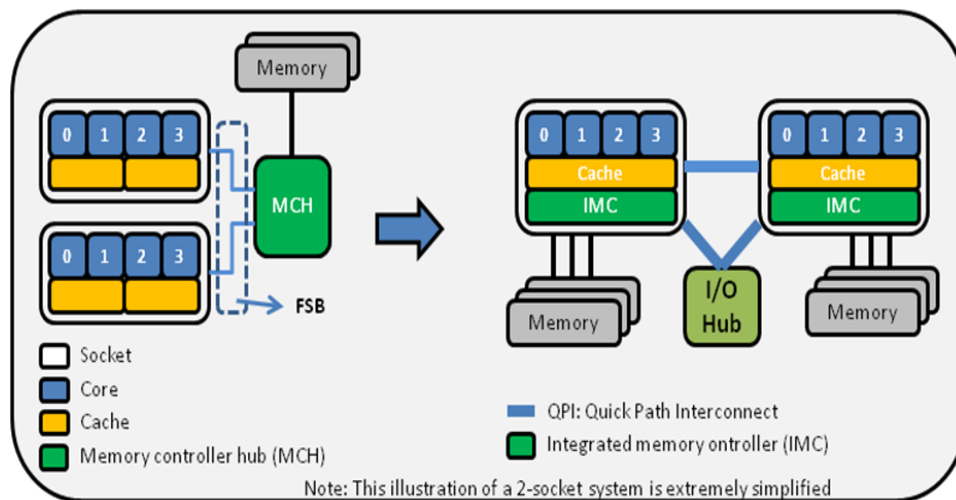


Figure 1. Transition from FSB to NUMA.

The clear symptom of bandwidth saturation for any parallel application is non-scaling behavior. In other words, an application that has saturated the available memory bandwidth will not scale effectively to more threads or cores. However, there are many causes for multi-threaded applications not to scale and some of these performance inhibiting factors include threading

overhead, synchronization overhead, load imbalance, and inappropriate granularity. Intel® Thread Profiler is designed to identify such performance issues at the application level.

The following results are taken after running the STREAM benchmark version 5.6 with various numbers of threads (only triad scores are shown).

	Function	Rate (MB/s)	Avg time	Min time	Max time
1 Thread	Triad:	7821.9511	0.0094	0.0092	0.0129
2 Threads	Triad:	8072.6533	0.0090	0.0089	0.0093
4 Threads	Triad:	7779.6354	0.0096	0.0093	0.0325

It is easy to see that STREAM does not benefit from having more threads on this particular platform (a single-socket Intel® Core™ 2 Quad-based system). Closer inspection of the results shows that even though there was a slight increase in the triad score for the two-thread version, the four-thread version performed even worse than the single threaded run.

Figure 2 shows Intel Thread Profiler analysis of the benchmark. The timeline view reveals that all threads are perfectly balanced and have no synchronization overheads. While it is a powerful tool for identifying threading performance issues at application level, Intel Thread Profiler will not detect memory bandwidth saturation in threaded applications.

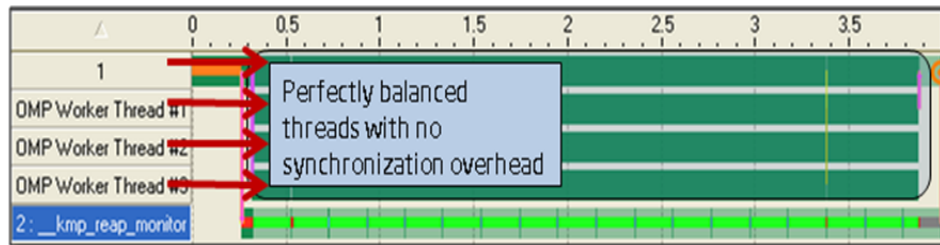


Figure 2. Intel Thread Profiler timeline view of the STREAM benchmark using four OpenMP* threads.

Advice

Intel® VTune™ Performance Analyzer and Performance Tuning Utility (PTU) used in combination with event-based sampling (EBS), can help developers measure application bandwidth usage, which can then be checked against the achievable (or theoretical) bandwidth on the system. Event-based sampling relies on the performance monitoring unit (PMU) supported by the processors.

VTune analyzer and PTU can help developers estimate the memory bandwidth usage of a particular application by using EBS. On Intel® Core™ 2 microarchitecture CPU_CLK_UNHALTED.CORE and BUS_TRANS_MEM.ALL_AGENTS performance events can be used to estimate the memory bandwidth.

- The CPU_CLK_UNHALTED.CORE event counts the number of core cycles while the core is not in a halt state. The core enters the halt state when it is running the HLT instruction.

- The BUS_TRANS_MEM.ALL_AGENTS event counts activity initiated by any agent on the bus. In systems where each processor is attached to a different bus, the count reflects only the activity for the bus on which the processor resides.

On Core 2-based systems memory bandwidth can be estimated by using the following formula:

$$(64 * \text{BUS_TRANS_MEM.ALL_AGENTS} * \text{CPU Frequency}) / \text{CPU_CLK_UNHALTED.CORE}$$

Module	Process	BUS_TRANS_ANY.ALL_AGENTS events	CPU_CLK_UNHALTED.CORE events
stream	stream	1,419,200,000	35,576,000,000

Figure 3. VTune analyzer EBS analysis of STREAM with four threads.

Figure 3 shows the EBS results of the STREAM benchmark when four threads were used. By using the above formula, it is possible to estimate the memory bandwidth usage of STREAM as 7.6Gb/sec.

$$\text{Memory Bandwidth} = (64 * 1,419,200,000 * 2.9\text{GHz}) / 35,576,000,000 = 7.6\text{GB/sec}$$

The STREAM-reported sustainable Triad score was 7.7GB/seconds, so the VTune analyzer-based calculation is quite reasonable. The STREAM benchmark was chosen to demonstrate how memory bandwidth measured using EBS can approximately measure the achievable memory bandwidth on a particular system.

If an application doesn't scale when more threads are added to take advantage of the available cores, and if Intel Thread Profiler doesn't show any application-level threading problems as mentioned above, then the following three steps can help the user determine whether or not a particular application is saturating the available memory bandwidth:

1. Run STREAM or similar benchmarks to get an idea of the sustainable memory bandwidth on the target system.
2. Run the target application under VTune analyzer or PTU and collect the appropriate performance counters using EBS. For Core 2 microarchitecture, these events are again CPU_CLK_UNHALTED.CORE and BUS_TRANS_MEM.ALL_AGENTS (Formula 1).
3. Compare VTune analyzer-measured memory bandwidth numbers to the sustainable or achievable memory bandwidth measured in step 1. If the application is saturating the available bandwidth, then this particular application won't scale with more cores.

Generally speaking, a memory-bound application (one whose performance is limited by the memory access speed) won't benefit from having more threads.

Usage Guidelines

The new Intel® Core™ i7 and Xeon® 5500 series processors are referred to as having an "uncore," which is that part of the processor that is external to all the individual cores. For example, the Intel Core i7 processor has four cores that share an L3 cache and a memory interface. The L3 and memory interface are considered to be part of the uncore (see Figure 4).

Neither the VTune analyzer nor PTU support the sampling of events that are triggered in the uncore of the processor, and the memory bandwidth measurement must be performed differently. The relevant performance events used for measuring bandwidth are not sampled using EBS as is usual with VTune analyzer or PTU; rather, they are counted using time-based sampling. This means that the bandwidth is measured for the entire system over a designated time range, and it isn't possible to see how much of the bandwidth usage comes from specific functions, processes, and modules.

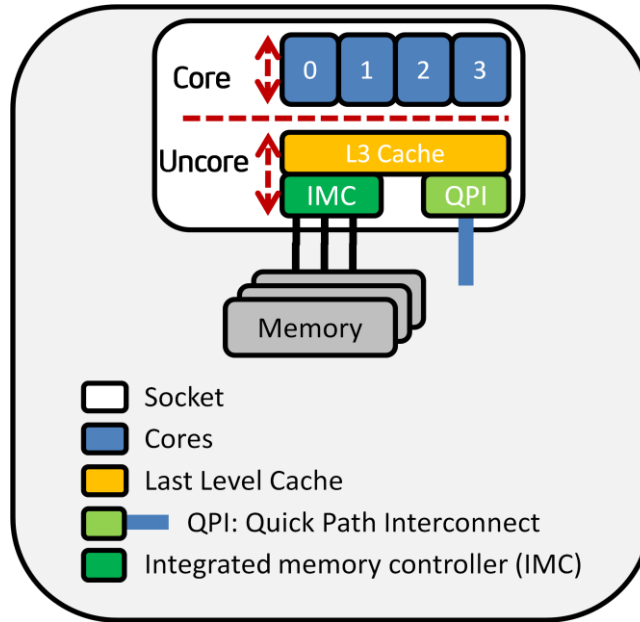


Figure 4. Simplified block diagram of a 4 core Nehalem processor.

The formula given above can be used to measure the memory bandwidth usage of any application, module, or function on Core 2 based systems except on Core 2 based Xeon MP processors, which also have uncore parts. The basic formula for measuring the memory bandwidth on Nehalem architecture-based systems can be given as follows:

$$\text{Memory Bandwidth} = 1.0\text{e-}9 * (\text{UNC_IMC_NORMAL_READS.ANY} + \text{UNC_IMC_WRITES.FULL.ANY}) * 64 / (\text{wall clock time in seconds})$$

Additional Resources

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

[Intel® VTune™ Performance Analyzer](#)

[STREAM: Sustainable Memory Bandwidth in High Performance Computers](#)

[Intel® Performance Tuning Utility](#)

[How Do I Measure Memory Bandwidth on an Intel® Core™ i7 or Intel® Xeon® Processor 5500 Series Platform Using Intel® VTune™ Performance Analyzer?](#)

Avoiding and Identifying False Sharing Among Threads

Abstract

In symmetric multiprocessor (SMP) systems, each processor has a local cache. The memory system must guarantee cache coherence. False sharing occurs when threads on different processors modify variables that reside on the same cache line. This invalidates the cache line and forces an update, which hurts performance. This article covers methods to detect and correct false sharing.

Background

False sharing is a well-known performance issue on SMP systems, where each processor has a local cache. It occurs when threads on different processors modify variables that reside on the same cache line, as illustrated in Figure 1. This circumstance is called false sharing because each thread is not actually sharing access to the same variable. Access to the same variable, or true sharing, would require programmatic synchronization constructs to ensure ordered data access.

The source line shown in red in the following example code causes false sharing:

```
double sum=0.0, sum_local[NUM_THREADS];
#pragma omp parallel num_threads(NUM_THREADS)
{
    int me = omp_get_thread_num();
    sum_local[me] = 0.0;

    #pragma omp for
    for (i = 0; i < N; i++)
        sum_local[me] += x[i] * y[i];

    #pragma omp atomic
    sum += sum_local[me];
}
```

There is a potential for false sharing on array `sum_local`. This array is dimensioned according to the number of threads and is small enough to fit in a single cache line. When executed in parallel, the threads modify different, but adjacent, elements of `sum_local` (the source line shown in red), which invalidates the cache line for all processors.

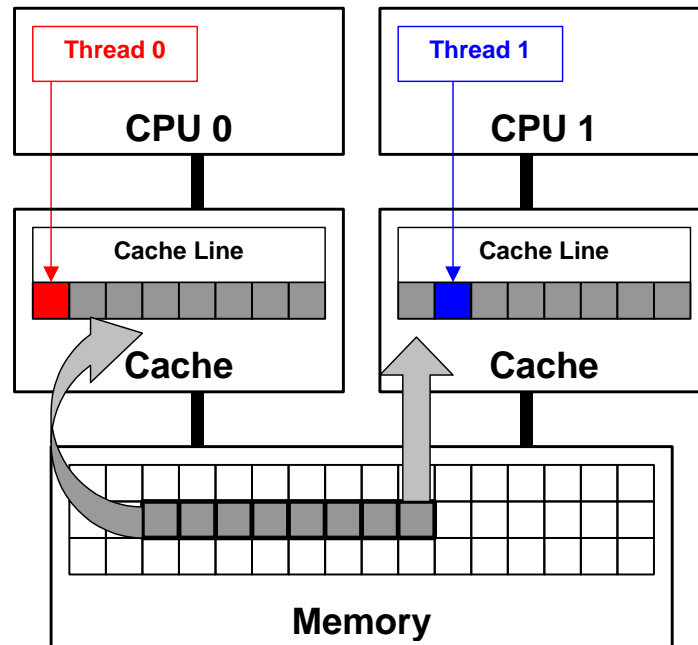


Figure 1. False sharing occurs when threads on different processors modify variables that reside on the same cache line. This invalidates the cache line and forces a memory update to maintain cache coherency.

In Figure 1, threads 0 and 1 require variables that are adjacent in memory and reside on the same cache line. The cache line is loaded into the caches of CPU 0 and CPU 1 (gray arrows). Even though the threads modify different variables (red and blue arrows), the cache line is invalidated, forcing a memory update to maintain cache coherency.

To ensure data consistency across multiple caches, multiprocessor-capable Intel® processors follow the MESI (Modified/Exclusive/Shared/Invalid) protocol. On first load of a cache line, the processor will mark the cache line as 'Exclusive' access. As long as the cache line is marked exclusive, subsequent loads are free to use the existing data in cache. If the processor sees the same cache line loaded by another processor on the bus, it marks the cache line with 'Shared' access. If the processor stores a cache line marked as 'S', the cache line is marked as 'Modified' and all other processors are sent an 'Invalid' cache line message. If the processor sees the same cache line which is now marked 'M' being accessed by another processor, the processor stores the cache line back to memory and marks its cache line as 'Shared'. The other processor that is accessing the same cache line incurs a cache miss.

The frequent coordination required between processors when cache lines are marked 'Invalid' requires cache lines to be written to memory and subsequently loaded. False sharing increases this coordination and can significantly degrade application performance.

Since compilers are aware of false sharing, they do a good job of eliminating instances where it could occur. For example, when the above code is compiled with optimization options, the compiler eliminates false sharing using thread-private temporal variables. Run-time false sharing from the above code will be only an issue if the code is compiled with optimization disabled.

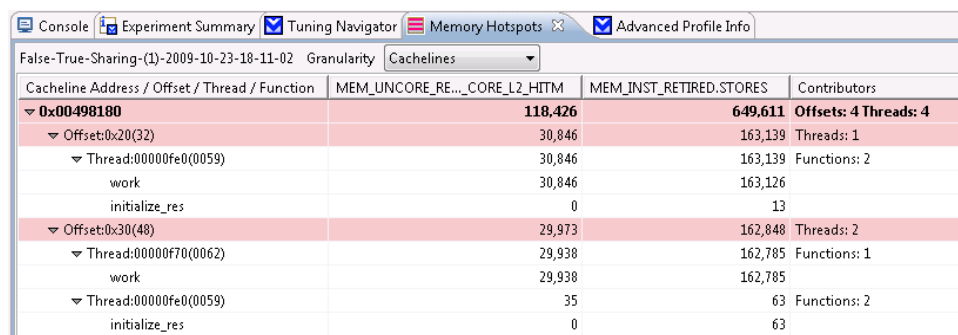
Advice

The primary means of avoiding false sharing is through code inspection. Instances where threads access global or dynamically allocated shared data structures are potential sources of false sharing. Note that false sharing can be obscured by the fact that threads may be accessing completely different global variables that happen to be relatively close together in memory. Thread-local storage or local variables can be ruled out as sources of false sharing.

The run-time detection method is to use the Intel® VTune™ Performance Analyzer or Intel® Performance Tuning Utility (Intel PTU, available at <http://software.intel.com/en-us/articles/intel-performance-tuning-utility/>). This method relies on event-based sampling that discovers places where cacheline sharing exposes performance visible effects. However, such effects don't distinguish between true and false sharing.

For systems based on the Intel® Core™ 2 processor, configure VTune analyzer or Intel PTU to sample the `MEM_LOAD_RETIRED.L2_LINE_MISS` and `EXT_SNOOP.ALL_AGENTS.HITM` events. For systems based on the Intel® Core i7 processor, configure to sample `MEM_UNCORE_RETIRED.OTHER_CORE_L2_HITM`. If you see a high occurrence of `EXT_SNOOP.ALL_AGENTS.HITM` events, such that it is a fraction of percent or more of `INST_RETIRED.ANY` events at some code regions on Intel® Core™ 2 processor family CPUs, or a high occurrence of `MEM_UNCORE_RETIRED.OTHER_CORE_L2_HITM` events on Intel® Core i7 processor family CPU, you have true or false sharing. Inspect the code of concentration of `MEM_LOAD_RETIRED.L2_LINE_MISS` and `MEM_UNCORE_RETIRED.OTHER_CORE_L2_HITM` events at the corresponding system at or near load/store instructions within threads to determine the likelihood that the memory locations reside on the same cache line and causing false sharing.

Intel PTU comes with predefined profile configurations to collect events that will help to locate false sharing. These configurations are "Intel® Core™ 2 processor family – Contested Usage" and "Intel® Core™ i7 processor family – False-True Sharing." Intel PTU Data Access analysis identifies false sharing candidates by monitoring different offsets of the same cacheline accessed by different threads. When you open the profiling results in Data Access View, the Memory Hotspot pane will have hints about false sharing at the cacheline granularity, as illustrated in Figure 2.



The screenshot shows the Intel PTU Memory Hotspots pane. The title bar includes 'Console', 'Experiment Summary', 'Tuning Navigator', 'Memory Hotspots', and 'Advanced Profile Info'. The main window title is 'False-True-Sharing-(1)-2009-10-23-18-11-02'. Below the title bar, there is a 'Granularity' dropdown menu set to 'Cachelines'. The table below displays memory hotspot data with columns: 'Cacheline Address / Offset / Thread / Function', 'MEM_UNCORE_RE..._CORE_L2_HITM', 'MEM_INST_RETIRED.STORES', and 'Contributors'.

Cacheline Address / Offset / Thread / Function	MEM_UNCORE_RE..._CORE_L2_HITM	MEM_INST_RETIRED.STORES	Contributors
▼ 0x00498180	118,426	649,611	Offsets: 4 Threads: 4
▼ Offset:0x20(32)	30,846	163,139	Threads: 1
▼ Thread:00000fe0(0059)	30,846	163,139	Functions: 2
work	30,846	163,126	
initialize_res	0	13	
▼ Offset:0x30(48)	29,973	162,848	Threads: 2
▼ Thread:00000f70(0062)	29,938	162,785	Functions: 1
work	29,938	162,785	
▼ Thread:00000fe0(0059)	35	63	Functions: 2
initialize_res	0	63	

Figure 2. False sharing shown in Intel PTU Memory Hotspots pane.

In Figure 2, memory offsets 32 and 48 (of the cacheline at address 0x00498180) were accessed by the ID=59 thread and the ID=62 thread at the work function. There is also some true sharing due to array initialization done by the ID=59 thread.

The pink color is used to hint about false sharing at a cacheline. Note the high figures for `MEM_UNCORE_RETIRED.OTHER_CORE_L2_HITM` associated with the cacheline and its corresponding offsets.

Once detected, there are several techniques to correct false sharing. The goal is to ensure that variables causing false sharing are spaced far enough apart in memory that they cannot reside on the same cache line. While the following is not an exhaustive list three possible methods are discussed below.

One technique is to use compiler directives to force individual variable alignment. The following source code demonstrates the compiler technique using `__declspec (align(n))` where `n` equals 64 (64 byte boundary) to align the individual variables on cache line boundaries.

```
__declspec (align(64)) int thread1_global_variable;
__declspec (align(64)) int thread2_global_variable;
```

When using an array of data structures, pad the structure to the end of a cache line to ensure that the array elements begin on a cache line boundary. If you cannot ensure that the array is aligned on a cache line boundary, pad the data structure to twice the size of a cache line. The following source code demonstrates padding a data structure to a cache line boundary and ensuring the array is also aligned using the compiler `__declspec (align(n))` statement where `n` equals 64 (64 byte boundary). If the array is dynamically allocated, you can increase the allocation size and adjust the pointer to align with a cache line boundary.

```
struct ThreadParams
{
    // For the following 4 variables: 4*4 = 16 bytes
    unsigned long thread_id;
    unsigned long v; // Frequent read/write access variable
    unsigned long start;
    unsigned long end;

    // expand to 64 bytes to avoid false-sharing
    // (4 unsigned long variables + 12 padding)*4 = 64
    int padding[12];
};

__declspec (align(64)) struct ThreadParams Array[10];
```

It is also possible to reduce the frequency of false sharing by using thread-local copies of data. The thread-local copy can be read and modified frequently and only when complete, copy the result back to the data structure. The following source code demonstrates using a local copy to avoid false sharing.

```
struct ThreadParams
{
    // For the following 4 variables: 4*4 = 16 bytes
    unsigned long thread_id;
    unsigned long v; // Frequent read/write access variable
    unsigned long start;
    unsigned long end;
};

void threadFunc(void *parameter)
```

```

{
    ThreadParams *p = (ThreadParams*) parameter;
    // local copy for read/write access variable
    unsigned long local_v = p->v;

    for(local_v = p->start; local_v < p->end; local_v++)
    {
        // Functional computation
    }

    p->v = local_v; // Update shared data structure only once
}

```

Usage Guidelines

Avoid false sharing but use these techniques sparingly. Overuse can hinder the effective use of the processor's available cache. Even with multiprocessor shared-cache designs, avoiding false sharing is recommended. The small potential gain for trying to maximize cache utilization on multi-processor shared cache designs does not generally outweigh the software maintenance costs required to support multiple code paths for different cache architectures.

Additional Resources

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

[Intel® VTune™ Performance Analyzer](#)

[Intel® Performance Tuning Utility](#)

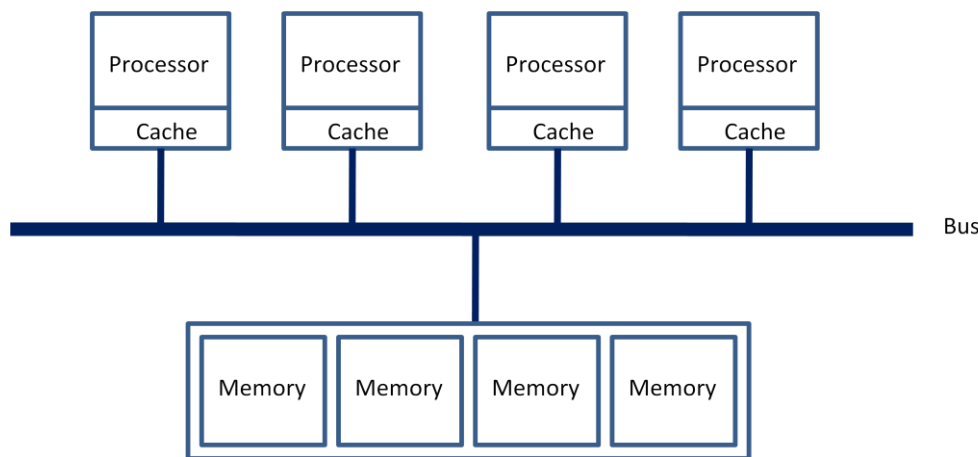
Optimizing Applications for NUMA

Abstract

NUMA, or **Non-Uniform Memory Access**, is a shared memory architecture that describes the placement of main memory modules with respect to processors in a multiprocessor system. Like most every other processor architectural feature, ignorance of NUMA can result in sub-par application memory performance. Fortunately, there are steps that can be taken to mitigate any NUMA-based performance issues or to even use NUMA architecture to the advantage of your parallel application. Such considerations include processor affinity, memory allocation using implicit operating system policies, and the use of the system APIs for assigning and migrating memory pages using explicit directives.

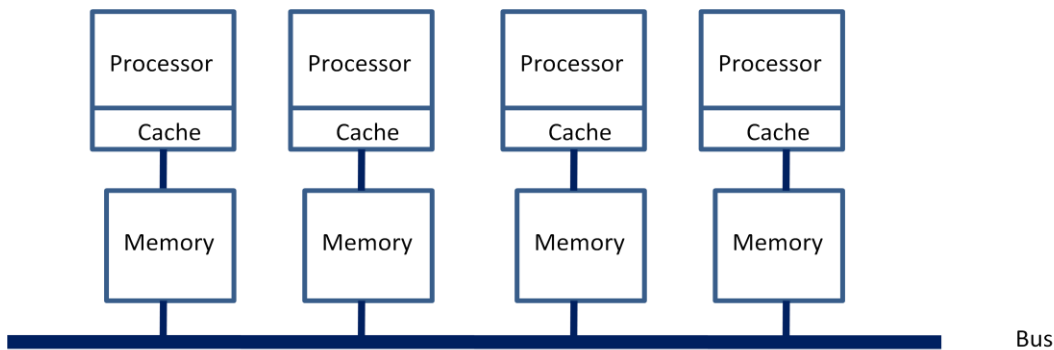
Background

Perhaps the best way to understand NUMA is to compare it with its cousin **UMA**, or **Uniform Memory Access**. In the UMA memory architecture, all processors access shared memory through a bus (or another type of interconnect) as seen in the following diagram:



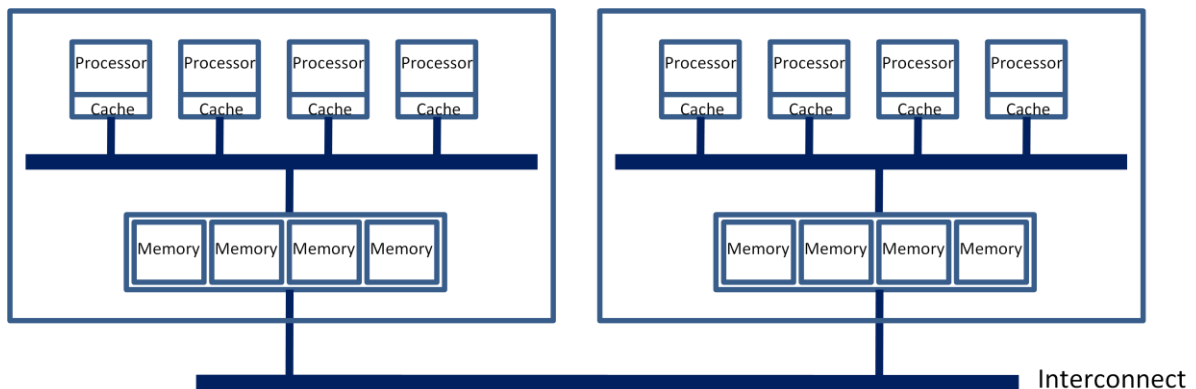
UMA gets its name from the fact that each processor must use the same shared bus to access memory, resulting in a memory access time that is uniform across all processors. Note that access time is also independent of data location within memory. That is, access time remains the same regardless of which shared memory module contains the data to be retrieved.

In the NUMA shared memory architecture, each processor has its own *local* memory module that it can access directly with a distinctive performance advantage. At the same time, it can also access any memory module belonging to another processor using a shared bus (or some other type of interconnect) as seen in the diagram below:



What gives NUMA its name is that memory access time varies with the location of the data to be accessed. If data resides in local memory, access is fast. If data resides in remote memory, access is slower. The advantage of the NUMA architecture as a *hierarchical* shared memory scheme is its potential to improve *average case access time* through the introduction of fast, local memory.

Modern multiprocessor systems mix these basic architectures as seen in the following diagram:



In this complex hierarchical scheme, processors are grouped by their physical location on one or the other multi-core CPU package or “node.” Processors within a node share access to memory modules as per the UMA shared memory architecture. At the same time, they may also access memory from the remote node using a shared interconnect, but with slower performance as per the NUMA shared memory architecture.

Advice

Two key notions in managing performance within the NUMA shared memory architecture are *processor affinity* and *data placement*.

Processor Affinity

Today’s complex operating systems assign application threads to processor cores using a scheduler. A scheduler will take into account system state and various policy objectives (e.g., “balance load across cores” or “aggregate threads on a few cores and put remaining cores to sleep”), then match application threads to physical cores accordingly. A given thread will execute on its assigned core for some period of time before being swapped out of the core to wait, as

other threads are given the chance to execute. If another core becomes available, the scheduler may choose to migrate the thread to ensure timely execution and meet its policy objectives.

Thread migration from one core to another poses a problem for the NUMA shared memory architecture because of the way it disassociates a thread from its local memory allocations. That is, a thread may allocate memory on node 1 at startup as it runs on a core within the node 1 package. But when the thread is later migrated to a core on node 2, the data stored earlier becomes remote and memory access time significantly increases.

Enter processor affinity. Processor affinity refers to the persistence of associating a thread/process with a particular processor resource instance, despite the availability of other instances. Using a system API, or by modifying an OS data structure (e.g., affinity mask), a specific core or set of cores can be associated with an application thread. The scheduler will then observe this affinity in its scheduling decisions for the lifetime of the thread. For example, a thread may be configured to run only on cores 0 through 3, all of which belong to quad core CPU package 0. The scheduler will choose among cores 0 through 3 without even considering migrating the thread to another package.

Exercising processor affinity ensures that memory allocations remain local to the thread(s) that need them. Several downsides, however, should be noted. In general, processor affinity may significantly harm system performance by restricting scheduler options and creating resource contention when better resources management could have otherwise been used. Besides preventing the scheduler from assigning waiting threads to unutilized cores, processor affinity restrictions may hurt the application itself when additional execution time on another node would have more than compensated for a slower memory access time.

Programmers must think carefully about whether processor affinity solutions are right for a particular application and shared system context. Note, finally, that processor affinity APIs offered by some systems support priority “hints” and affinity “suggestions” to the scheduler in addition to explicit directives. Using such suggestions, rather than forcing unequivocal structure on thread placement, may ensure optimal performance in the common case yet avoid constraining scheduling options during periods of high resource contention.

Data Placement Using Implicit Memory Allocation Policies

In the simple case, many operating systems transparently provide support for NUMA-friendly data placement. When a single-threaded application allocates memory, the processor will simply assign memory pages to the physical memory associated with the requesting thread’s node (CPU package), thus ensuring that it is local to the thread and access performance is optimal.

Alternatively, some operating systems will wait for the first memory access before committing on memory page assignment. To understand the advantage here, consider a multithreaded application with a start-up sequence that includes memory allocations by a main control thread, followed by the creation of various worker threads, followed by a long period of application processing or service. While it may seem reasonable to place memory pages local to the allocating thread, in fact, they are more effectively placed local to the worker threads that will access the data. As such, the operating system will observe the first access request and commit page assignments based on the requester’s node location.

These two policies (local to first access and local to first request) together illustrate the importance of an application programmer being aware of the NUMA context of the program’s deployment. If the page placement policy is based on first access, the programmer can exploit

this fact by including a carefully designed data access sequence at startup that will generate “hints” to the operating system on optimal memory placement. If the page placement policy is based on requester location, the programmer should ensure that memory allocations are made by the thread that will subsequently access the data and not by an initialization or control thread designed to act as a provisioning agent.

Multiple threads accessing the same data are best co-located on the same node so that the memory allocations of one, placed local to the node, can benefit all. This may, for example, be used by prefetching schemes designed to improve application performance by generating data requests in advance of actual need. Such threads must generate data placement that is local to the actual consumer threads for the NUMA architecture to provide its characteristic performance speedup.

It should be noted that when an operating system has fully consumed the physical memory resources of one node, memory requests coming from threads on the same node will typically be fulfilled by sub-optimal allocations made on a remote node. The implication for memory-hungry applications is to correctly size the memory needs of a particular thread and to ensure local placement with respect to the accessing thread.

For situations where a large number of threads will randomly share the same pool of data from all nodes, the recommendation is to stripe the data evenly across all nodes. Doing so spreads the memory access load and avoids bottleneck access patterns on a single node within the system.

Data Placement Using Explicit Memory Allocation Directives

Another approach to data placement in NUMA-based systems is to make use of system APIs that explicitly configure the location of memory page allocations. An example of such APIs is the `libnuma` library for Linux.

Using the API, a programmer may be able to associate virtual memory address ranges with particular nodes, or simply to indicate the desired node within the memory allocation system call itself. With this capability, an application programmer can ensure the placement of a particular data set regardless of which thread allocates it or which thread accesses it first. This may be useful, for example, in schemes where complex applications make use of a memory management thread acting on behalf of worker threads. Or, it may prove useful for applications that create many short-lived threads, each of which have predictable data requirements. Pre-fetching schemes are another area that could benefit considerably from such control.

The downside of this scheme, of course, is the management burden placed on the application programmer in handling memory allocations and data placement. Misplaced data may cause performance that is significantly worse than default system behavior. Explicit memory management also presupposes fine-grained control over processor affinity throughout application use.

Another capability available to the application programmer through NUMA-based memory management APIs is memory page migration. In general, migration of memory pages from one node to another is an expensive operation and something to be avoided. Having said this, given an application that is both long-lived and memory intensive, migrating memory pages to re-establish a NUMA-friendly configuration may be worth the price.³ Consider, for example, a long-lived application with various threads that have terminated and new threads that have been created but reside on another node. Data is now no longer local to the threads that need it and

sub-optimal access requests now dominate. Application-specific knowledge of a thread's lifetime and data needs can be used to determine whether an explicit migration is in order.

Usage Guidelines

The key issue in determining whether the performance benefits of the NUMA architecture can be realized is ***data placement***. The more often that data can effectively be placed in memory local to the processor that needs it, the more overall access time will benefit from the architecture. By providing each node with its own local memory, memory accesses can avoid throughput limitations and contention issues associated with a shared memory bus. In fact, memory constrained systems can theoretically improve their performance by up to the number of nodes on the system by virtue of accessing memory in a fully parallelized manner.

Conversely, the more data fails to be local to the node that will access it, the more memory performance will suffer from the architecture. In the NUMA model, the time required to retrieve data from an adjacent node within the NUMA model will be significantly higher than that required to access local memory. In general, as the *distance* from a processor increases, the cost of accessing memory increases.

Additional Resources

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

Drepper, Ulrich. "What Every Programmer Should Know About Memory". November 2007.

Intel® 64 and IA-32 Architectures Optimization Reference Manual. See Section 8.8 on "Affinities and Managing Shared Platform Resources". March 2009.

Lameter, Christoph. "Local and Remote Memory: Memory in a Linux/NUMA System". June 2006.

Automatic Parallelization with Intel® Compilers

Abstract

Multithreading an application to improve performance can be a time-consuming activity. For applications where most of the computation is carried out in simple loops, the Intel® compilers may be able to generate a multithreaded version automatically.

In addition to high-level code optimizations, the Intel Compilers also enable threading through automatic parallelization and [OpenMP*](#) support. With automatic parallelization, the compiler detects loops that can be safely and efficiently executed in parallel and generates multithreaded code. OpenMP allows programmers to express parallelism using compiler directives and C/C++ pragmas.

Background

The Intel® C++ and Fortran Compilers have the ability to analyze the dataflow in loops to determine which loops can be safely and efficiently executed in parallel. Automatic parallelization can sometimes result in shorter execution times on multicore systems. It also relieves the programmer from:

- Searching for loops that are good candidates for parallel execution
- Performing dataflow analysis to verify correct parallel execution
- Adding parallel compiler directives manually.

Adding the `-Qparallel` (Windows*) or `-parallel` (Linux* or Mac OS* X) option to the compile command is the only action required of the programmer. However, successful parallelization is subject to certain conditions that are described in the next section.

The following Fortran program contains a loop with a high iteration count:

```
PROGRAM TEST
PARAMETER (N=10000000)
REAL A, C(N)
DO I = 1, N
  A = 2 * I - 1
  C(I) = SQRT(A)
ENDDO
PRINT*, N, C(1), C(N)
END
```

Dataflow analysis confirms that the loop does not contain data dependencies. The compiler will generate code that divides the iterations as evenly as possible among the threads at runtime. The number of threads defaults to the total number of processor cores (which may be greater than the number of physical cores if Intel® Hyper Threading Technology is enabled), but may be set independently via the `OMP_NUM_THREADS` environment variable. The parallel speed-up for a given loop depends on the amount of work, the load balance among threads, the overhead of thread creation and synchronization, etc., but it will generally be less than linear relative to the number of threads used. For a whole program, speed-up depends on the ratio of parallel to serial computation (see any good textbook on parallel computing for a description of Amdahl's Law).

Advice

Three requirements must be met for the compiler to parallelize a loop. First, the number of iterations must be known before entry into a loop so that the work can be divided in advance. A while-loop, for example, usually cannot be made parallel. Second, there can be no jumps into or out of the loop. Third, and most important, the loop iterations must be independent. In other words, correct results must not logically depend on the order in which the iterations are executed. There may, however, be slight variations in the accumulated rounding error, as, for example, when the same quantities are added in a different order. In some cases, such as summing an array or other uses of temporary scalars, the compiler may be able to remove an apparent dependency by a simple transformation.

Potential aliasing of pointers or array references is another common impediment to safe parallelization. Two pointers are aliased if both point to the same memory location. The compiler may not be able to determine whether two pointers or array references point to the same memory location, for example, if they depend on function arguments, run-time data, or the results of complex calculations. If the compiler cannot prove that pointers or array references are safe and that iterations are independent, it will not parallelize the loop, except in limited cases when it is deemed worthwhile to generate alternative code paths to test explicitly for aliasing at run-time. If the programmer knows that parallelization of a particular loop is safe, and that potential aliases can be ignored, this fact can be communicated to the compiler with a C pragma (`#pragma parallel`) or Fortran directive (`!DIR$ PARALLEL`). An alternative way in C to assert that a pointer is not aliased is to use the `restrict` keyword in the pointer declaration, along with the `-Qrestrict` (Windows) or `-restrict` (Linux or Mac OS* X) command-line option. However, the compiler will never parallelize a loop that it can prove to be unsafe.

The compiler can only effectively analyze loops with a relatively simple structure. For example, it cannot determine the thread-safety of a loop containing external function calls because it does not know whether the function call has side effects that introduce dependences. Fortran 90 programmers can use the `PURE` attribute to assert that subroutines and functions contain no side effects. Another way, in C or Fortran, is to invoke inter-procedural optimization with the `-Qipo` (Windows) or `-ipo` (Linux or Mac OS X) compiler option. This gives the compiler the opportunity to inline or analyze the called function for side effects.

When the compiler is unable to automatically parallelize complex loops that the programmer knows could safely be executed in parallel, OpenMP is the preferred solution. The programmer typically understands the code better than the compiler and can express parallelism at a coarser granularity. On the other hand, automatic parallelization can be effective for nested loops, such as those in a matrix multiply. Moderately coarse-grained parallelism results from threading of the outer loop, allowing the inner loops to be optimized for fine grained parallelism using vectorization or software pipelining.

Just because a loop can be parallelized does not mean that it should be parallelized. The compiler uses a cost model with a threshold parameter to decide whether to parallelize a loop. The `-Qpar-threshold[n]` (Windows) and `-par-threshold[n]` (Linux) compiler options adjust this parameter. The value of `n` ranges from 0 to 100, where 0 means to always parallelize a safe loop, irrespective of the cost model, and 100 tells the compiler to only parallelize those loops for which a performance gain is highly probable. The default value of `n` is conservatively set to 100; sometimes, reducing the threshold to 99 may result in a significant increase in the number of loops parallelized. The pragma `#parallel always` (`!DIR$ PARALLEL ALWAYS` in Fortran) may be used to override the cost model for an individual loop.

The switches `-Qpar-report[n]` (Windows) or `-par-report[n]` (Linux), where `n` is 1 to 3, show which loops were parallelized. Look for messages such as:

```
test.f90(6) : (col. 0) remark: LOOP WAS AUTO-PARALLELIZED
```

The compiler will also report which loops could not be parallelized and the reason why, as in the following example:

```
serial loop: line 6
flow data dependence from line 7 to line 8, due to "c"
```

This is illustrated by the following example:

```
void add (int k, float *a, float *b)
{
  for (int i = 1; i < 10000; i++)
    a[i] = a[i+k] + b[i];
}
```

The compile command `'icl -c -Qparallel -Qpar-report3 add.cpp'` results in messages such as the following:

```
procedure: add
test.c(7): (col. 1) remark: parallel dependence: assumed ANTI
dependence between a line 7 and a line 7. flow data dependence
assumed
...
test.c(7): (col. 1) remark: parallel dependence: assumed FLOW
dependence between a line 7 and b line 7.
```

Because the compiler does not know the value of `k`, it must assume that the iterations depend on each other, as for example if `k` equals -1. However, the programmer may know otherwise, due to specific knowledge of the application (e.g., `k` always greater than 10000), and can override the compiler by inserting a pragma:

```
void add (int k, float *a, float *b)
{
  #pragma parallel
  for (int i = 1; i < 10000; i++)
    a[i] = a[i+k] + b[i];
}
```

The messages now show that the loop is parallelized:

```
procedure: add
test.c(6): (col. 1) remark: LOOP WAS AUTO-PARALLELIZED.
```

However, it is now the programmer's responsibility not to call this function with a value of `k` that is less than 10000, to avoid possible incorrect results.

Usage Guidelines

Try building the computationally intensive kernel of your application with the `-parallel` (Linux or Mac OS X) or `-Qparallel` (Windows) compiler switch. Enable reporting with `-par-report3`

(Linux) or `-Qpar-report3` (Windows) to find out which loops were parallelized and which loops could not be parallelized. For the latter, try to remove data dependencies and/or help the compiler disambiguate potentially aliased memory references. Compiling at `-O3` enables additional high-level loop optimizations (such as loop fusion) that may sometimes help autparallelization. Such additional optimizations are reported in the compiler optimization report generated with `-opt-report-phase hlo`. Always measure performance with and without parallelization to verify that a useful speedup is being achieved.

If `-openmp` and `-parallel` are both specified on the same command line, the compiler will only attempt to parallelize those loops that do not contain OpenMP directives. For builds with separate compiling and linking steps, be sure to link the OpenMP runtime library when using automatic parallelization. The easiest way to do this is to use the compiler driver for linking, by means, for example, of `icl -Qparallel` (Windows) or `ifort -parallel` (Linux or Mac OS X). On Mac OS X systems, you may need to set the `DYLD_LIBRARY_PATH` environment variable within Xcode to ensure that the OpenMP dynamic library is found at runtime.

Additional Resources

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

“Optimizing Applications/Using Parallelism: Automatic Parallelization” in the Intel® C++ Compiler User and Reference Guides or The Intel® Fortran Compiler User and Reference Guides

[Efficient Exploitation of Parallelism on Pentium® III and Pentium® 4 Processor-Based Systems](#)

[An Overview of the Parallelization Implementation Methods in Intel® Parallel Composer](#)

Parallelism in the Intel® Math Kernel Library

Abstract

Software libraries provide a simple way to get immediate performance benefits on multicore, multiprocessor, and cluster computing systems. The Intel® Math Kernel Library (Intel® MKL) contains a large collection of functions that can benefit math-intensive applications. This chapter will describe how Intel MKL can help programmers achieve superb serial and parallel performance in common application areas. This material is applicable to IA-32 and Intel® 64 processors on Windows*, Linux*, and Mac OS* X operating systems.

Background

Optimal performance on modern multicore and multiprocessor systems is typically attained only when opportunities for parallelism are well exploited and the memory characteristics underlying the architecture are expertly managed. Sequential codes must rely heavily on instruction and register level SIMD parallelism and cache blocking to achieve best performance. Threaded programs must employ advanced blocking strategies to ensure that multiple cores and processors are efficiently used and the parallel tasks evenly distributed. In some instances, out-of-core implementations can be used to deal with large problems that do not fit in memory.

Advice

One of the easiest ways to add parallelism to a math-intensive application is to use a threaded, optimized library. Not only will this save the programmer a substantial amount of development time, it will also reduce the amount of test and evaluation effort required. Standardized APIs also help to make the resulting code more portable.

Intel MKL provides a comprehensive set of math functions that are optimized and threaded to exploit all the features of the latest Intel® processors. The first time a function from the library is called, a runtime check is performed to identify the hardware on which the program is running. Based on this check, a code path is chosen to maximize use of instruction- and register level SIMD parallelism and to choose the best cache-blocking strategy. Intel MKL is also designed to be threadsafe, which means that its functions operate correctly when simultaneously called from multiple application threads.

Intel MKL is built using the Intel® C++ and Fortran Compilers and threaded using OpenMP*. Its algorithms are constructed to balance data and tasks for efficient use of multiple cores and processors. The following table shows the math domains that contain threaded functions (this information is based on Intel MKL 10.2 Update 3):

Linear Algebra	Used in applications from finite-element analysis engineering codes to modern animation
BLAS (Basic Linear Algebra Subprograms)	All matrix-matrix operations (level 3) are threaded for both dense and sparse BLAS. Many vector-vector (level 1) and matrix-vector (level 2) operations are threaded for dense matrices in 64-bit programs running on the Intel® 64 architecture. For sparse matrices, all level 2 operations except for the sparse triangular solvers are threaded.

LAPACK (Linear Algebra Package)	Several computational routines are threaded from each of the following types of problems: linear equation solvers, orthogonal factorization, singular value decomposition, and symmetric eigenvalue problems. LAPACK also calls BLAS, so even non-threaded functions may run in parallel.
ScaLAPACK (Scalable LAPACK)	A distributed-memory parallel version of LAPACK intended for clusters.
PARDISO	This parallel direct sparse solver is threaded in its three stages: reordering (optional), factorization, and solve (if solving with multiple right-hand sides).
Fast Fourier Transforms	Used for signal processing and applications that range from oil exploration to medical imaging
Threaded FFTs (Fast Fourier Transforms)	Threaded with the exception of 1D real and split-complex FFTs.
Cluster FFTs	Distributed-memory parallel FFTs intended for clusters.
Vector Math	Used in many financial codes
VML (Vector Math Library)	Arithmetic, trigonometric, exponential/logarithmic, rounding, etc.

Because there is some overhead involved in the creation and management of threads, it is not always worthwhile to use multiple threads. Consequently, Intel MKL does not create threads for small problems. The size that is considered small is relative to the domain and function. For level 3 BLAS functions, threading may occur for a dimension as small as 20, whereas level 1 BLAS and VML functions will not thread for vectors much smaller than 1000.

Intel MKL should run on a single thread when called from a threaded region of an application to avoid over-subscription of system resources. For applications that are threaded using OpenMP, this should happen automatically. If other means are used to thread the application, Intel MKL behavior should be set using the controls described below. In cases where the library is used sequentially from multiple threads, Intel MKL may have functionality that can be helpful. As an example, the Vector Statistical Library (VSL) provides a set of vectorized random number generators that are not threaded, but which offer a means of dividing a stream of random numbers among application threads. The `SkipAheadStream()` function divides a random number stream into separate blocks, one for each thread. The `LeapFrogStream()` function will divide a stream so that each thread gets a subsequence of the original stream. For example, to divide a stream between two threads, the Leapfrog method would provide numbers with odd indices to one thread and even indices to the other.

Performance

Figure 1 provides an example of the kind of performance a user could expect from DGEMM, the double precision, general matrix-matrix multiply function included in Intel MKL. This BLAS function plays an important role in the performance of many applications. The graph shows the

performance in Gflops for a variety of rectangular sizes. It demonstrates how performance scales across processors (speedups of up to 1.9x on two threads, 3.8x on four threads, and 7.9x on eight threads), as well as achieving nearly 94.3% of peak performance at 96.5 Gflops.

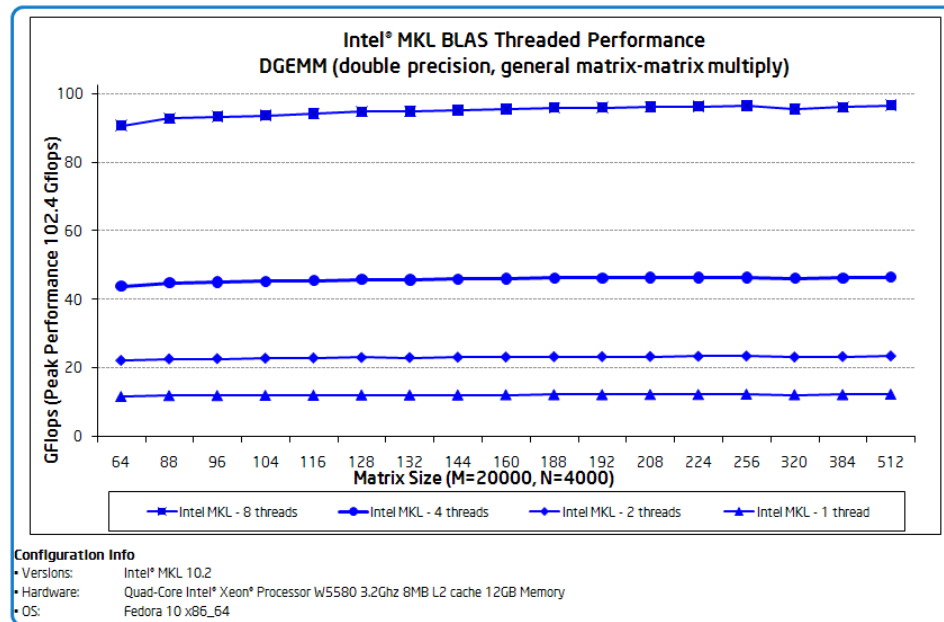


Figure 1. Performance and scalability of the BLAS matrix-matrix multiply function.

Usage Guidelines

Since Intel MKL is threaded using OpenMP, its behavior can be affected by OpenMP controls. For added control over threading behavior, Intel MKL provides a number of service functions that mirror the OpenMP controls. These functions allow the user to control the number of threads the library uses, either as a whole or per domain (i.e., separate controls for BLAS, LAPACK, etc.). One application of these independent controls is the ability to allow nested parallelism. For example, behavior of an application threaded using OpenMP could be set using the `OMP_NUM_THREADS` environment variable or `omp_set_num_threads()` function, while Intel MKL threading behavior was set independently using the Intel MKL specific controls: `MKL_NUM_THREADS` or `mkl_set_num_threads()` as appropriate. Finally, for those who must always run Intel MKL functions on a single thread, a sequential library is provided that is free of all dependencies on the threading runtime.

Intel® Hyper-Threading Technology is most effective when each thread performs different types of operations and there are under-utilized resources on the processor. However, Intel MKL fits neither of these criteria, because the threaded portions of the library execute at high efficiency using most of the available resources and perform identical operations on each thread. Because of that, Intel MKL will by default use only as many threads as there are physical cores.

Additional Resources

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

[Intel® Math Kernel Library](#)

[Netlib: Information about BLAS, LAPACK, and ScaLAPACK](#)

Threading and Intel® Integrated Performance Primitives

Abstract

There is no universal threading solution that works for all applications. Likewise, there are multiple ways for applications built with Intel® Integrated Performance Primitives (Intel® IPP) to utilize multiple threads of execution. Threading can be implemented with at the low primitive level (within the Intel IPP library) or at the high operating system level. This chapter will describe some of the ways in which an application that utilizes Intel IPP can safely and successfully take advantage of multiple threads of execution.

Background

Intel IPP is a collection of highly optimized functions for digital media and data-processing applications. The library includes optimized functions for frequently used fundamental algorithms found in a variety of domains, including signal processing, image, audio, and video encode/decode, data compression, string processing, and encryption. The library takes advantage of the extensive SIMD (single instruction multiple data) and SSE (streaming SIMD extensions) instruction sets and multiple hardware execution threads available in modern Intel® processors. Many of the SSE instructions found in today's processors are modeled after those on DSPs (digital signal processors) and are ideal for optimizing algorithms that operate on arrays and vectors of data.

The Intel IPP library is available for applications built for the Windows*, Linux*, Mac OS* X, QNX*, and VxWorks* operating systems. It is compatible with the Intel® C and Fortran Compilers, the Microsoft Visual Studio* C/C++ compilers, and the gcc compilers included with most Linux distributions. The library has been validated for use with multiple generations of Intel and compatible AMD processors, including the Intel® Core™ and Intel® Atom™ processors. Both 32-bit and 64-bit operating systems and architectures are supported.

Introduction

The Intel IPP library has been constructed to accommodate a variety of approaches to multithreading. The library is thread-safe by design, meaning that the library functions can be safely called from multiple threads within a single application. Additionally, variants of the library are provided with multithreading built in, by way of the Intel OpenMP* library, giving you an immediate performance boost without requiring that your application be rewritten as a multithreaded application.

The Intel IPP primitives (the low-level functions that comprise the base Intel IPP library) are a collection of algorithmic elements designed to operate repetitively on data vectors and arrays, an ideal condition for the implementation of multithreaded applications. The primitives are independent of the underlying operating system; they do not utilize locks, semaphores, or global variables, and they rely only on the standard C library memory allocation routines (`malloc/realloc/calloc/free`) for temporary and state memory storage. To further reduce dependency on external system functions, you can use the `i_malloc` interface to substitute your own memory allocation routines for the standard C routines.

In addition to the low-level algorithmic primitives, the Intel IPP library includes a collection of industry-standard, high-level applications and tools that implement image, media and speech codecs (encoders and decoders), data compression libraries, string processing functions, and cryptography tools. Many of these high-level tools use multiple threads to divide the work between two or more hardware threads.

Even within a singled-threaded application, the Intel IPP library provides a significant performance boost by providing easy access to SIMD instructions (MMX, SSE, etc.) through a set of functions designed to meet the needs of numerically intensive algorithms.

Figure 1 shows relative average performance improvements measured for the various Intel IPP product domains, as compared to the equivalent functions implemented without the aid of MMX/SSE instructions. Your actual performance improvement will vary.

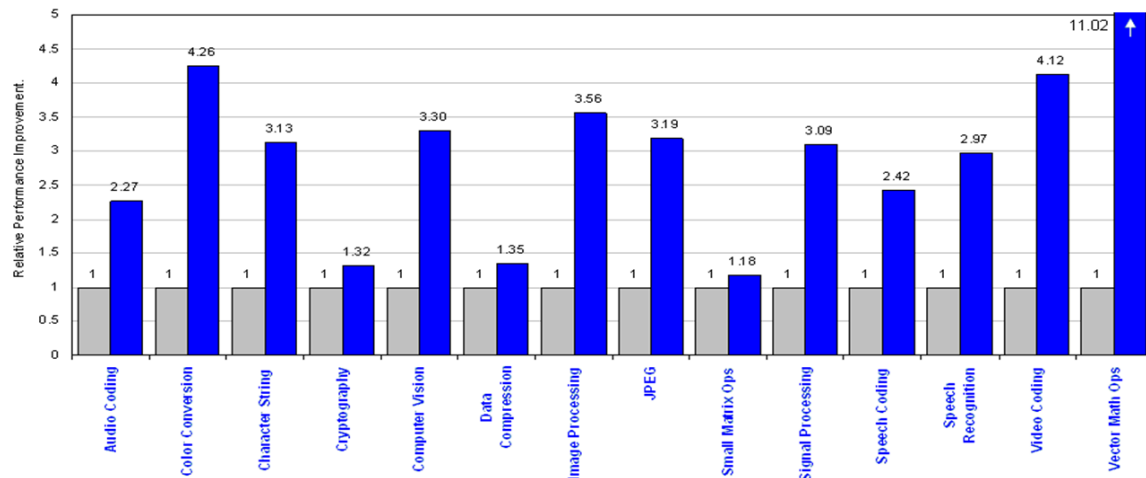


Figure 1. Relative performance improvements for various Intel® IPP product domains.

System configuration: Intel® Xeon® Quad-Core Processor, 2.8GHz, 2GB using Windows® XP and the Intel IPP 6.0 library

Advice

The simplest and quickest way to take advantage of multiple hardware threads with the Intel IPP library is to use a multi-threaded variant of the library or to incorporate one of the many threaded sample applications provided with the library. This requires no significant code rework and can provide additional performance improvements beyond those that result simply from the use of Intel IPP.

Three fundamental variants of the library are delivered (as of version 6.1): a single-threaded static library; a multithreaded static library; and a multithreaded dynamic library. All three variants of the library are thread-safe. The single-threaded static library should be used in kernel-mode applications or in those cases where the use of the OpenMP library either cannot be tolerated or is not supported (as may be the case with a real-time operating system).

Two Threading Choices: OpenMP Threading and Intel IPP

The low-level primitives within Intel IPP are basic atomic operations, which limits the amount of parallelism that can be exploited to approximately 15% of the library functions. The Intel OpenMP library has been utilized to implement this "behind the scenes" parallelism and is enabled by default when using one of the multi-threaded variants of the library.

A complete list of the multi-threaded primitives is provided in the `ThreadedFunctionsList.txt` file located in the Intel IPP documentation directory.

Note: the fact that the Intel IPP library is built with the Intel C compiler and OpenMP does not require that applications must also be built using these tools. The Intel IPP library primitives are delivered in a binary format compatible with the C compiler for the relevant operating system (OS) platform and are ready to link with applications. Programmers can build applications that use Intel IPP application with Intel tools or other preferred development tools.

Controlling OpenMP Threading in the Intel IPP Primitives

The default number of OpenMP threads used by the threaded Intel IPP primitives is equal to the number of hardware threads in the system, which is determined by the number and type of CPUs present. For example, a quad-core processor that supports Intel® Hyper-Threading Technology (Intel® HT Technology) has eight hardware threads (each of four cores supports two threads). A dual-core CPU that does not include Intel HT Technology has two hardware threads.

Two Intel IPP primitives provide universal control and status regarding OpenMP threading within the multi-threaded variants of the Intel IPP library: `ippSetNumThreads()` and `ippGetNumThreads()`. Call `ippGetNumThreads` to determine the current thread cap and use `ippSetNumThreads` to change the thread cap. `ippSetNumThreads` will not allow you to set the thread cap beyond the number of available hardware threads. This thread cap serves as an upper bound on the number of OpenMP software threads that can be used within a multi-threaded primitive. Some Intel IPP functions may use fewer threads than specified by the thread cap in order to achieve their optimum parallel efficiency, but they will never use more than the thread cap allows.

To disable OpenMP within a threaded variant of the Intel IPP library, call `ippSetNumThreads(1)` near the beginning of the application, or link the application with the Intel IPP single-threaded static library.

The OpenMP library references several configuration environment variables. In particular, `OMP_NUM_THREADS` sets the default number of threads (the thread cap) to be used by the OpenMP library at run time. However, the Intel IPP library will override this setting by limiting the number of OpenMP threads used by an application to be either the number of hardware threads in the system, as described above, or the value specified by a call to `ippSetNumThreads`, whichever is lower. OpenMP applications that do not use the Intel IPP library may still be affected by the `OMP_NUM_THREADS` environment variable. Likewise, such OpenMP applications are not affected by a call to the `ippSetNumThreads` function within Intel IPP applications.

Nested OpenMP

If an Intel IPP application also implements multithreading using OpenMP, the threaded Intel IPP primitives the application calls may execute as single-threaded primitives. This happens if the Intel IPP primitive is called within an OpenMP parallelized section of code and if nested parallelization has been disabled (which is the default case) within the Intel OpenMP library.

Nesting of parallel OpenMP regions risks creating a large number of threads that effectively oversubscribe the number of hardware threads available. Creating parallel region always incurs overhead, and the overhead associated with the nesting of parallel OpenMP regions may outweigh the benefit.

In general, OpenMP threaded applications that use the Intel IPP primitives should disable multi-threading within the Intel IPP library either by calling `ippSetNumThreads(1)` or by using the single-threaded static Intel IPP library.

Core Affinity

Some of the Intel IPP primitives in the signal-processing domain are designed to execute parallel threads that exploit a merged L2 cache. These functions (single and double precision FFT, Div, Sqrt, etc.) need a shared cache to achieve their maximum multi-threaded performance. In other words, the threads within these primitives should execute on cores located on a single die with a shared cache. To ensure that this condition is met, the following OpenMP environment variable should be set before an application using the Intel IPP library runs:

```
KMP_AFFINITY=compact
```

On processors with two or more cores on a single die, this condition is satisfied automatically and the environment variable is superfluous. However, for those systems with more than two dies (e.g., a Pentium® D processor or a multi-socket motherboard), where the cache serving each die is not shared, failing to set this OpenMP environmental variable may result in significant performance degradation for this class of multi-threaded Intel IPP primitives.

Usage Guidelines

Threading Within an Intel IPP Application

Many multithreaded examples of applications that use the Intel IPP primitives are provided as part of the Intel IPP library. Source code is included with all of these samples. Some of the examples implement threading at the application level, and some use the threading built into the Intel IPP library. In most cases, the performance gain due to multithreading is substantial.

When using the primitives in a multithreaded application, disabling the Intel IPP library's built-in threading is recommended, using any of the techniques described in the previous section. Doing so ensures that there is no competition between the built-in threading of the library and the application's threading mechanism, helping to avoid an oversubscription of software threads to the available hardware threads.

Most of the library primitives emphasize operations on arrays of data, because the Intel IPP library takes advantage of the processor's SIMD instructions, which are well suited to vector operations. Threading is natural for operations on multiple data elements that are largely independent. In general, the easiest way to thread with the library is by using data decomposition, or splitting large blocks of data into smaller blocks and working on those smaller blocks with multiple identical parallel threads of execution.

Memory and Cache Alignment

When working with large blocks of data, improperly aligned data will typically reduce throughput. The library includes a set of memory allocation and alignment functions to address this issue. Additionally, most compilers can be configured to pad structures to ensure bus-efficient alignment of data.

Cache alignment and the spacing of data relative to cache lines is very important when implementing parallel threads. This is especially true for parallel threads that contain constructs of looping Intel IPP primitives. If the operations of multiple parallel threads frequently utilize

coincident or shared data structures, the write operations of one thread may invalidate the cache lines associated with the data structures of a “neighboring” thread.

When building parallel threads of identical Intel IPP operations (data decomposition), be sure to consider the relative spacing of the decomposed data blocks being operated on by the parallel threads *and* the spacing of any control data structures used by the primitives within those threads. Take especial care when the control structures hold state information that is updated on each iteration of the Intel IPP functions. If these control structures share a cache line, an update to one control structure may invalidate a neighboring structure.

The simplest solution is to allocate these control data structures so they occupy a multiple of the processor’s cache line size (typically 64 bytes). Developers can also use the compiler’s align operators to insure these structures, and arrays of such structures, always align with cache line boundaries. Any wasted bytes used to pad control structures will more than make up for the lost bus cycles required to refresh a cache line on each iteration of the primitives.

Pipelined Processing with DMIP

In an ideal world, applications would adapt at run-time to optimize their use of the SIMD instructions available, the number of hardware threads present, and the size of the high-speed cache. Optimum use of these three key resources might achieve near perfect parallel operation of the application, which is the essential aim behind the DMIP library that is part of Intel IPP.

The DMIP approach to parallelization, building parallel sequences of Intel IPP primitives that are executed on cache-optimal sized data blocks, enables application performance gains of several factors over those that operate sequentially over an entire data set with each function call.

For example, rather than operate over an entire image, break the image into cacheable segments and perform multiple operations on each segment, while it remains in the cache. The sequence of operations is a calculation pipeline and is applied to each tile until the entire data set is processed. Multiple pipelines running in parallel can then be built to amplify the performance.

To find out more detail about this technique, see “[A Landmark in Image Processing: DMIP.](#)”

Threaded Performance Results

Additional high-level threaded library tools included with Intel IPP offer significant performance improvements when used in multicore environments.

For example, the Intel IPP data compression library provides drop-in compatibility for the popular ZLIB, BZIP2, GZIP and LZO lossless data-compression libraries. The Intel IPP versions of the BZIP2 and GZIP libraries take advantage of a multithreading environment by using native threads to divide large files into many multiple blocks to be compressed in parallel, or by processing multiple files in separate threads. Using this technique, the GZIP library is able to achieve as much as a 10x performance gain on a quad-core processor, when compared to an equivalent single-threaded implementation without Intel IPP.

In the area of multi-media (e.g., video and image processing), an Intel IPP version of H.264 and VC 1 decoding is able to achieve near theoretical maximum scaling to the available hardware threads by using multiple native threads to parallelize decoding, reconstruction, and de-blocking operations on the video frames. Executing this Intel IPP enhanced H.264 decoder on a quad-core processor results in performance gains of between 3x and 4x for a high-definition bit stream.

Final Remarks

There is no single approach that is guaranteed to provide the best performance for all situations. The performance improvements achieved through the use of threading and the Intel IPP library depend on the nature of the application (e.g., how easily it can be threaded), the specific mix of Intel IPP primitives it can use (threaded versus non-threaded primitives, frequency of use, etc.), and the hardware platform on which it runs (number of cores, memory bandwidth, cache size and type, etc.).

Additional Resources

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

Taylor, Stewart. *Optimizing Applications for Multi-Core Processors: Using the Intel® Integrated Performance Primitives*, Second Edition. Intel Press, 2007.

User's Guide, Intel® Integrated Performance Primitives for Windows OS on IA-32 Architecture*, Document Number 318254-007US, March 2009.

Reference Manual, Intel® Integrated Performance Primitives for Intel® Architecture: Deferred Mode Image Processing Library, Document Number: 319226-002US, January 2009.

Intel IPP i_malloc sample code, located in advanced-usage samples in linkage section

[Wikipedia article on OpenMP*](#)

[OpenMP.org](#)

[A Landmark in Image Processing: DMIP](#)

Using Intel® Inspector XE 2011 to Find Data Races in Multithreaded Code

Abstract

[Intel® Inspector XE 2011](#), one of the three components within the [Intel® Parallel Studio XE](#) suite product, is used to find correctness errors in Windows or Linux applications. Intel Inspector XE 2011 automatically finds memory errors, deadlocks and other conditions that could lead to deadlocks, data races, thread stalls, and more.

Background

Debugging threaded applications can be difficult, because debuggers change the runtime performance, which can mask race conditions. Even print statements can mask issues, because they use synchronization and operating system functions. Consider the code sample below from the perspective of what threading errors may lie dormant.

```
color col;

static color_t render_one_pixel (
    int x,
    int y,
    unsigned int *local_mbox,
    volatile unsigned int &serial,
    int startx,
    int stopx,
    int starty,
    int stopy)
{
    ...

    col=trace(&primary);

    ...

    R=(int) (col.r*255);
    ...
}
```

Let's take a close look at some common threading errors. In this code snippet the global variable `col` is modified in function `render_one_pixel`. It is clear that if multiple threads are writing to variable `col` the value in `col` will be dependent on which thread writes the value last. This is a classic example of a data race.

Race conditions are tricky to detect because, in a given instance, the variables might "win the race" in the order that happens to make the program function correctly. Just because a program works once doesn't mean that it will always work. Testing your program on various machines, some with Hyper-Threading Technology and some with multiple physical processors, is one approach but cumbersome and unpredictable due to problem reproduction during testing. Tools such as the Intel Inspector XE 2011 can help. Traditional debuggers may not be useful for detecting race conditions because they cause one thread to stop the "race" while the other

threads can continue and may significantly change the runtime behavior thereby obscuring the data race.

Advice

Use Intel Inspector XE 2011 to facilitate debugging of multithreaded applications. Intel Inspector XE 2011 provides very valuable parallel execution information and debugging hints. Using dynamic binary instrumentation, Intel Inspector XE 2011 executes your application and monitors common threading APIs, and all memory accesses in an attempt to identify coding errors. It can find the infrequent errors that never seem to happen during testing but always seem to happen at a customer's site readily. These are called intermittent bugs and are unique to multithreaded programming. The tool is designed to detect and locate such notorious errors. The important thing to remember when using the tool is to try to exercise all code paths while accessing the least amount of memory possible, this will speed up the data-collection process. Usually, a small change to the source code or data set is required to reduce the amount of data processed by the application.

To prepare a program for Intel Inspector XE 2011 analysis, compile with optimization disabled and debugging symbols enabled. Launch the standalone Intel Inspector XE 2011 GUI from the Windows "Start" menu. Create a new Project and specify the application to analyze and the directory for the application to run in. Click on the "New Analysis" icon on the toolbar. Select the 3rd level: "Locate Deadlocks and Dataraces" under the "Threading Error Analysis". Click "Start" to start the analysis.

After clicking "Start", Intel Inspector XE 2001 runs the application using dynamic binary instrumentation. Once the application exits, the tool presents a summary of its findings

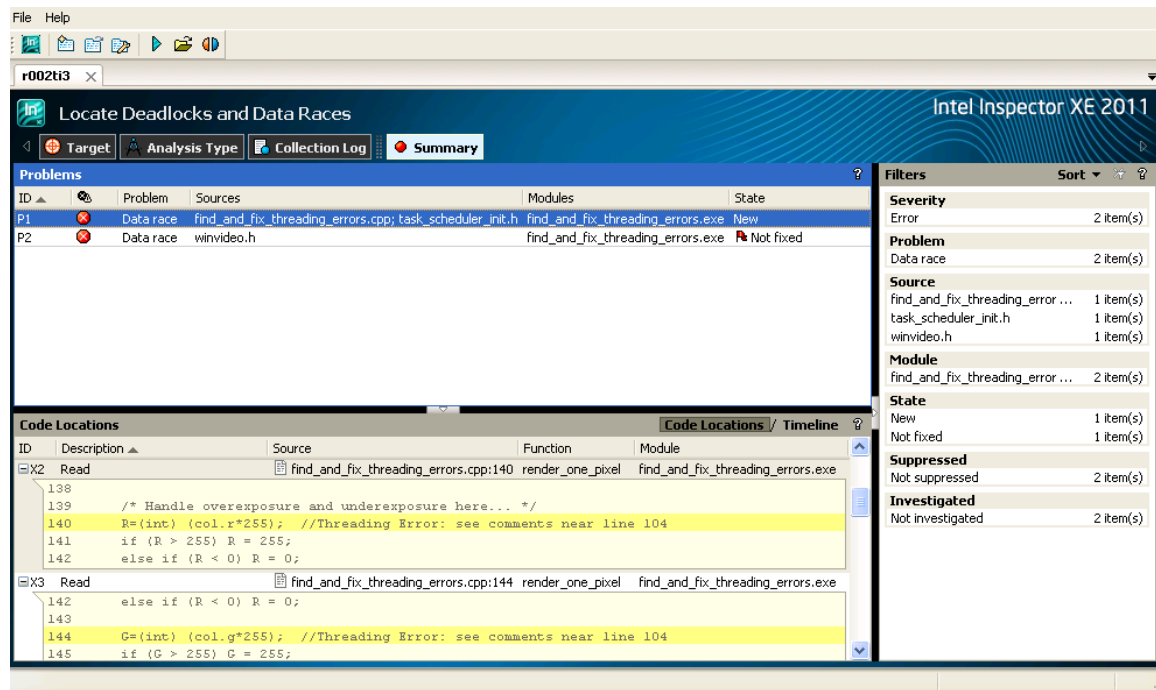


Figure 1. Summary overview in Intel Inspector XE 2011

For further problem investigation Intel Inspector XE 2011 provides detailed description of errors with the associated function call stacks and productivity features such as filtering and state management. A quick reference to the source code helps to identify the code range where the error was detected. By double clicking a problem in the Summary list it displays a view of the error at the source level (Figure 2).

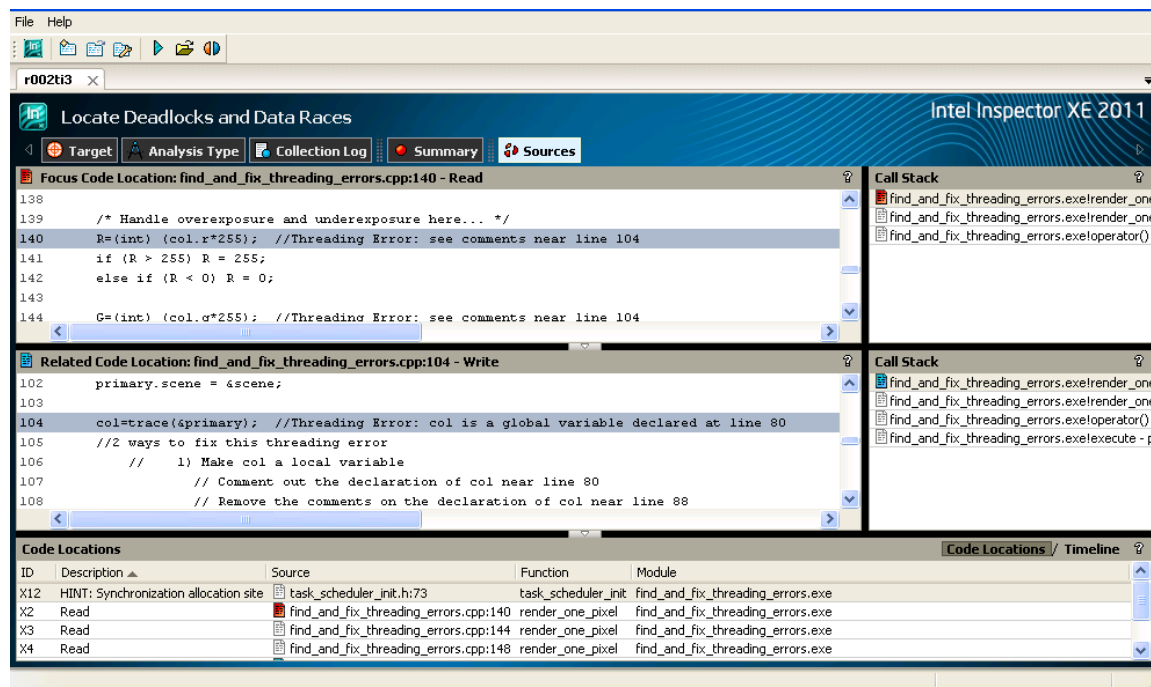


Figure 2. Source view in the Intel Inspector XE 2011

Once the error report is obtained and the root cause is identified with the help of Intel Inspector XE 2011, a developer should consider approaches how to fix the problems. Below is discussed the general considerations of avoiding data race conditions in parallel code and given the advice of how to fix the problem in the examined code.

Modify code to use a local variable instead of a global

In the code sample the variable `col` declared at line #80 could be declared as a local declared at line #88. (see the comments provided in the sample) If each thread is no longer referencing global data then there is no race condition, because each thread will have its own copy of the variable. This is the preferred method for fixing this issue.

Use a mutex to control access to global data

There are many algorithmic reasons for accessing global data, and it may not be possible to convert the global variable to a local variable. In these cases controlling access to the global by using a mutex is how threads typically achieve safe global data access.

This sample example happens to use Intel® Threading Building Blocks (Intel® TBB) to create and manage threads but Intel Inspector XE 2011 also supports numerous other threading models. Intel TBB provides several mutex patterns that can be used to control access to global data.

```

#include <tbb/mutex.h>
tbb::mutex countMutex;

static color_t render_one_pixel (
    int x,
    int y,
    unsigned int *local_mbox,
    volatile unsigned int &serial,
    int startx,
    int stopx,
    int starty,
    int stopy)
{
    tbb::mutex::scoped_lock lock(countMutex);

    col=trace(&primary);

    ...

    R=(int) (col.r*255);
    ...
}

```

In this new snippet, a `countMutex` is declared as a `scoped_lock`. The semantics of a `scoped_lock` are as follows: the lock is acquired by the `scoped_lock` constructor and released by the destructor automatically when the code leaves the block. Therefore only one thread is allowed to execute `render_one_pixel()` at a time, if additional threads call `render_one_pixel()` they will hit the `scoped_lock` and be forced to wait until the previous thread completes. The use of a mutex does affect performance and it is critical to make the scope of a mutex be as small as possible so that threads wait for the shortest interval.

Use a concurrent container to control access to global data

In addition to using a mutex to control access to global data, Intel TBB also provides several highly concurrent container classes. A concurrent container allows multiple threads to access and update data in the container. Containers provided by Intel TBB offer a higher level of concurrency by using 2 methods: fine grain locking, lock free algorithms. The use of these containers does come with an additional overhead and tradeoffs do need to be considered with regards to whether the concurrency speedup makes up for this overhead.

Usage Guidelines

Intel Inspector XE 2011 currently is available for the 32-bit and 64-bit versions of the Microsoft* Windows XP, Windows Vista, and Windows 7 operating systems and integrates into Microsoft* Visual Studio 2005, 2008 and 2010, it is also available on 32-bit and 64-bit versions of Linux*.

Note that Intel Inspector XE 2011 performs dynamic analysis, not static analysis. Intel Inspector XE 2011 only analyzes code that is executed. Therefore, multiple analyses exercising different parts of the program may be necessary to ensure adequate code coverage.

Intel Inspector XE 2011 instrumentation increases the CPU and memory requirements of an application so choosing a small but representative test problem is very important. Workloads with

runtimes of a few seconds are best. Workloads do not have to be realistic. They just have to exercise the relevant sections of multithreaded code.

Additional Resources

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

[Intel Parallel Inspector XE 2011](#)

Curing Thread Imbalance Using Intel® Parallel Amplifier

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.

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.

```
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).

Line	Source	CPU Time
116		
117	for(i=start; i<end; i++) {	
118	for(j=0; j<i-1; j++) {	0.041s
119	distx = pow((x[0][j] - x[0][i]), 2);	
120	disty = pow((x[1][j] - x[1][i]), 2);	7.047s
121	distz = pow((x[2][j] - x[2][i]), 2);	
122	dist = sqrt(distx + disty + distz);	15.250s
123	lPot += 1.0 / dist;	0.994s
124	}	
125	}	
126	gPot[tid] = lPot;	
127	}	

Figure 1. Source code view of the hotspot analysis.

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.

Function		Module		CPU Time by Utilization	
- Bottom-up Tree				Poor Ok Ideal Over	
computePot		potential_tpool.exe	23.334s		
tPoolComputePot		potential_tpool.exe	4.025s		
_Cipow		MSVCR90D.dll	4.387s		
_CIsqrt		MSVCR90D.dll	1.065s		
_Cipow		potential_tpool.exe	0.232s		
_CIsqrt		potential_tpool.exe	0.063s		
updatePositions		potential_tpool.exe	0.004s		

Figure 2. Concurrency analysis results.

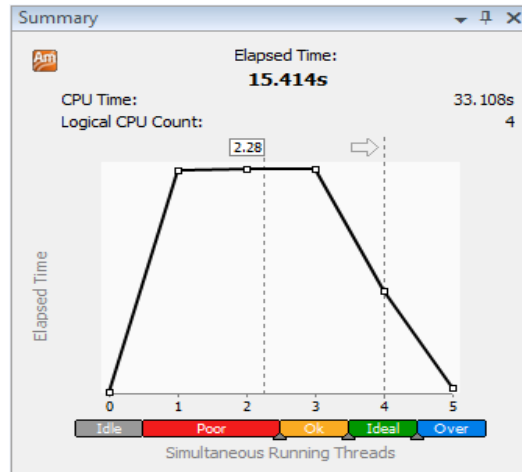






Figure 3. Summary view of the concurrency results.

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.

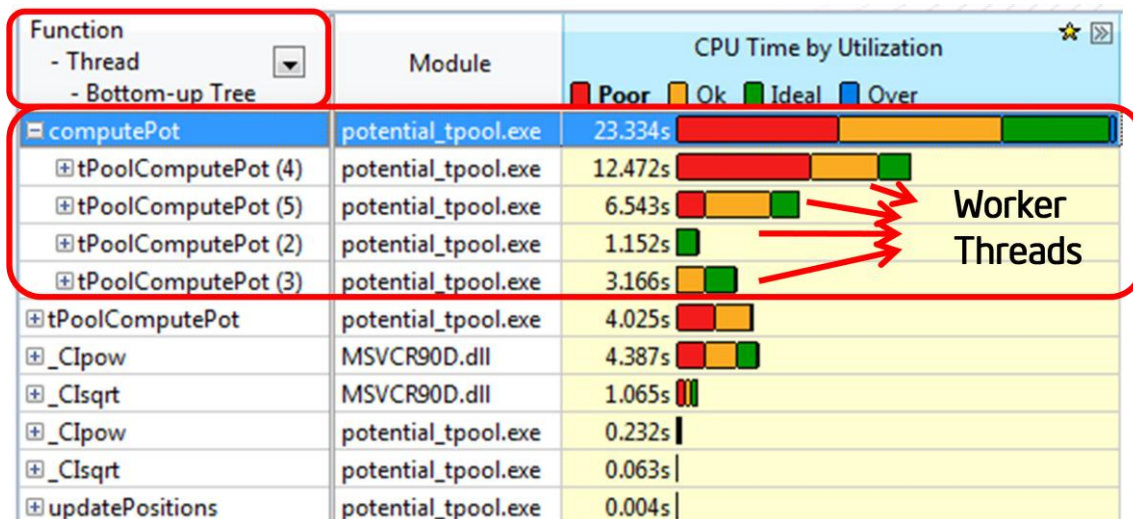


Figure 4. Concurrency analysis results showing Function->Thread grouping.

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.

```
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).

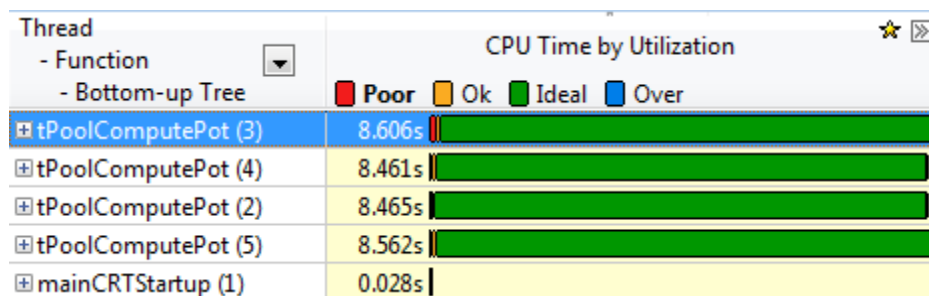


Figure 5. Concurrency analysis results after the change.

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%.

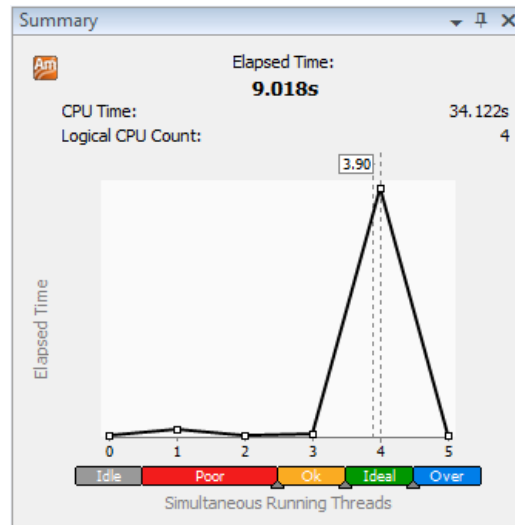


Figure 6. Summary view of the load-balanced version.

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](#)

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

Getting Code Ready for Parallel Execution with Intel® Parallel Composer

Abstract

Developers have the choice among a number of approaches for introducing parallelism into their code. This article provides an overview of the methods available in Intel® Parallel Composer, along with a comparison of their key benefits. While Intel Parallel Composer covers development in C/C++ on Windows* only, many of the methods also apply (with the suitable compiler) to development in Fortran and/or on Linux*.

Background

While the Intel® Compilers have some ways of automatically detecting and optimizing suitable code constructs for parallel execution (e.g., vectorization and auto/parallelization), most of the methods require code modifications. The pragmas or functions inserted rely on runtime libraries that actually perform the decomposition and schedule execution of the parallel threads, such as OpenMP*, Intel® Threading Building Blocks (Intel® TBB), and the Win32* API. The main difference among the approaches is the level of control they provide over execution details; in general, the more control provided, the more intrusive the required code changes are.

Parallelization using OpenMP*

OpenMP is an industry standard for portable multi-threaded application development. The Intel® C++ Compiler supports the OpenMP C/C++ version 3.0 API specification available at the OpenMP web site (<http://www.openmp.org>). Parallelism with OpenMP is controllable by the user through the use of OpenMP directives. This approach is effective at fine-grain (loop-level) and large-grain (function-level) threading. OpenMP directives provide an easy and powerful way to convert serial applications into parallel applications, enabling potentially big performance gains from parallel execution on multi-core systems. The directives are enabled with the `/Qopenmp` compiler option and will be ignored without the compiler option. This characteristic allows building both the serial and parallel versions of the application from the same source code. For shared memory parallel computers, it also allows for simple comparisons between serial and parallel runs.

The following table shows commonly used OpenMP* directives:

Directive	Description
<pre>#pragma omp parallel for [clause] ... for - loop</pre>	Parallelizes the loop that immediately follows the pragma.
<pre>#pragma omp parallel sections [clause] ... { [#pragma omp section structured-block] ... }</pre>	Distributes the execution of the different sections among the threads in the parallel team. Each structured block is executed once by one of the threads in the team in the context of its implicit task.
<pre>#pragma omp master structured-block</pre>	The code contained within the master construct is executed by the master thread in the thread team.

<code>#pragma omp critical [(name)] structured-block</code>	Provides mutual exclusion access to the structured-block. Only one critical section is allowed to execute at one time anywhere in the program.
<code>#pragma omp barrier</code>	Used to synchronize the execution of multiple threads within a parallel region. Ensures all the code occurring before the barrier has been completed by all the threads, before any thread can execute any of the code past the barrier directive.
<code>#pragma omp atomic expression-statement</code>	Provides mutual exclusion via hardware synchronization primitives. While a critical section provides mutual exclusion access to a block of code, the <code>atomic</code> directive provides mutual access to a single assignment statement.
<code>#pragma omp threadprivate (list)</code>	Specifies a list of global variables being replicated, one instance per thread (i.e., each thread works on an individual copy of the variable).

Example 1:

```
void sp_1a(float a[], float b[], int n) {
    int i;
    #pragma omp parallel shared(a,b,n) private(i)
    {
        #pragma omp for
        for (i = 0; i < n; i++)

            a[i] = 1.0 / a[i];

        #pragma omp single
        a[0] = a[0] * 10;
        #pragma omp for nowait
        for (i = 0; i < n; i++)
            b[i] = b[i] / a[i];
    }
}
icl /c /Qopenmp par1.cpp
par2.cpp(5): (col. 5) remark: OpenMP DEFINED LOOP WAS PARALLELIZED.
par2.cpp(10): (col. 5) remark: OpenMP DEFINED LOOP WAS PARALLELIZED.
par2.cpp(3): (col. 3) remark: OpenMP DEFINED REGION WAS PARALLELIZED.
```

The `/Qopenmp-report[n]` compiler option, where `n` is a number between 0 and 2, can be used to control the OpenmMP parallelizer's level of diagnostic messages. Use of this option requires the programmer to specify the `/Qopenmp` option. If `n` is not specified, the default is `/Qopenmp-report1` which displays diagnostic messages indicating loops, regions, and sections successfully parallelized.

Because only directives are inserted into the code, it is possible to make incremental code changes. The ability to make incremental code changes helps programmers maintain serial consistency. When the code is run on one processor, it gives the same result as the unmodified source code. OpenMP is a single source code solution that supports multiple platforms and operating systems. There is also no need to determine the number of cores, because the OpenMP runtime chooses the right number automatically.

OpenMP version 3.0 contains a new task-level parallelism construct that simplifies parallelizing functions, in addition to the loop-level parallelism for which OpenMP is most commonly used. The tasking model allows parallelizing programs with irregular patterns of dynamic data structures or with complicated control structures like recursion that are hard to parallelize efficiently. The task pragmas operate in the context of a parallel region and create explicit tasks. When a task pragma is encountered lexically within a parallel region, the code inside the task block is conceptually queued to be executed by one of the threads executing the parallel region. To preserve sequential semantics, all tasks queued within the parallel region are guaranteed to complete by the end of the parallel region. The programmer is responsible for ensuring that no dependencies exist and that dependencies are appropriately synchronized between explicit tasks, as well as between code inside and outside explicit tasks.

Example 2:

```
#pragma omp parallel
#pragma omp single
{
    for(int i = 0; i < size; i++)
    {
        #pragma omp task
        setQueen (new int[size], 0, i, myid);
    }
}
```

Intel® C++ Compiler Language Extensions for Parallelism

The Intel® Compiler uses simple C and C++ language extensions to make parallel programming easier. There are four keywords introduced within this version of the compiler:

- `__taskcomplete`
- `__task`
- `__par`
- `__critical`

In order for the application to benefit from the parallelism afforded by these keywords, the compiler switch `/Qopenmp` must be used during compilation. The compiler will link in the appropriate runtime support libraries, and the runtime system will manage the actual degree of parallelism. The parallel extensions utilize the OpenMP 3.0 runtime library but abstract out the use of the OpenMP pragmas and directives, keeping the code more naturally written in C or C++. The mapping between the parallelism extensions and the OpenMP constructs are as follows:

Parallel Extension	OpenMP
--------------------	--------

<code>__par</code>	<code>#pragma omp parallel for</code>
<code>__critical</code>	<code>#pragma omp critical</code>
<code>__taskcomplete S1</code>	<code>#pragma omp parallel</code> <code>#pragma omp single</code> <code>{ S1 }</code>
<code>__task S2</code>	<code>#pragma omp task</code> <code>{ S2 }</code>

The keywords are used as statement prefixes.

Example 3:

```
__par for (i = 0; i < size; i++)
  setSize (new int[size], 0, i)

__taskcomplete {
  __task sum(500, a, b, c);
  __task sum(500, a+500, b+500, c+500)
}

if ( !found )
  __critical item_count++;
```

Intel® Cilk™ Plus

Intel Cilk Plus language extensions included in the Intel® C++ Compiler add fine-grained task support to C and C++, making it easy to add parallelism to both new and existing software to efficiently exploit multiple processors. Intel Cilk Plus is made up of these main features:

- Set of keywords (`cilk_spawn`, `cilk_sync`, and `cilk_for`), for expression of task parallelism.
- Reducers, which eliminate contention for shared variables among tasks by automatically creating views of them for each task and reducing them back to a shared value after task completion.
- Array notations, which enable data parallelism of whole functions or operations which can then be applied to whole or parts of arrays or scalars.
- The `simd` pragma, which lets you express vector parallelism for utilizing hardware SIMD parallelism while writing standard compliant C/C++ code with an Intel compiler.
- Elemental function, which can be invoked either on scalar arguments or on array elements in parallel. You define an elemental function by adding "`__declspec(vector)`" (on Windows*) and "`__attribute__((vector))`" (on Linux*) before the function signature.

Item	Description
------	-------------

<code>cilk_spawn</code>	Keyword that modifies a function call statement to tell the runtime system that the function may (but is not required to) run in parallel with the caller.
<code>cilk_sync</code>	Keyword that indicates the current function cannot continue past this point until its spawned children have returned.
<code>cilk_for</code>	Keyword used to specify that loop iterations are permitted to run in parallel; is a replacement for the normal C/C++ for loop. This statement divides a loop into chunks containing one or more loop iterations. Each chunk is executed serially, and is spawned as a chunk during the execution of the loop.
<code>cilk_grainsize</code>	Pragma used to specify the grain (chunk) size for one <code>cilk_for</code> loop.
<code>CILK_NWORKERS</code>	Environment variable used to specify the number of worker threads.

Example: cilk_spawn, cilk_sync

In the example below, `cilk_spawn` does not spawn anything nor create any new threads. It indicates to the Cilk Plus runtime that a free worker thread can steal the code following the call to `fib(n-1)` and do it in parallel with the function call that follows.

```
x = cilk_spawn fib(n-1);
y = fib(n-2);
cilk_sync;
return x+y;
```

Example: cilk_for, cilk Reducer

In the example below, `cilk_for` causes multiple instances of the code in the loop body to be spawned into execution cores and executed in parallel. The Reducer avoids a data race and allows the reduction operation to execute without explicit locking.

```
cilk::reducer_opadd< double > sum(0.0);
cilk_for(i = 1; i <= n; ++i) {
    double h = sqrt(1-i/n) * (i/n);
    sum = sum + h*w;
}
return sum.get_value();
```

Example: Array Notation

In the example below, the usual subscript syntax in C/C++ is replaced by an array section descriptor to achieve the same result as looping over each element of the arrays. The difference between using the array notation and the standard C/C++ loop is that here, there is no serial ordering implied. Therefore, the expected code generation strategy from the compiler is vector parallelism, i.e., use the SSE instructions to implement the additions in a SIMD fashion. The compiler generates vector, SIMD code for operations on arrays including all the language built-in

operators, such as '+', '*', '&', '&&', etc.

```
A[:] = B[:] + C[:];
```

Example: pragma simd

Vectorization using `#pragma simd` instructs the compiler to enforce vectorization of loops. It is designed to minimize the amount of source code changes needed in order to obtain vectorized code. The SIMD pragma can be used to vectorize loops that the compiler does not normally auto-vectorize even with the use of vectorization hints such as `"pragma vector always"` or `"pragma ivdep"`.

```
char foo(char *A, int n){
    int i;
    char x = 0;

    #ifdef SIMD
    #pragma simd reduction(+:x)
    #endif
    #ifdef IVDEP
    #pragma ivdep
    #endif
    for (i = 0; i < n; ++i){
        x = x + A[i];
    }
    return x;
}
```

The following are compilations of the above code with various flags used. The results of the compilation commands are shown to illustrate how the SIMD pragma can affect the vectorization of the simple loop.

```

>icl /c /Qvec-report2 simd.cpp

simd.cpp
simd.cpp(12) (col. 3): remark: loop was not vectorized: existence of
vector dependence.

>icl /c /Qvec-report2 simd.cpp /DIVDEP

simd.cpp
simd.cpp(12) (col. 3): remark: loop was not vectorized: existence of
vector dependence.

>icl /c /Qvec-report2 simd.cpp /DSIMD

simd.cpp
simd.cpp(12) (col. 3): remark: SIMD LOOP WAS VECTORIZED.

```

Example: Elemental function

An elemental function is a regular function, which can be invoked either on scalar arguments or on array elements in parallel.

```

__declspec(vector)
int vfun_add_one(int x)
{
    return x+1;
}

```

There are three ways that the above function can be invoked. Each of these is illustrated below. The first calls the function from the body of a for-loop on individual elements of an array. This invokes the function in a scalar fashion. The second uses a `cilk_for` loop to parallelize the execution of loop iterations. The third uses the Array Notation feature of Intel Cilk Plus to execute the elemental function on corresponding elements from the arrays and stores the results in corresponding elements of the result array.

```
for (int i = 0; i < length; ++i)
    res1[i] = vfun_add_one(a[i]);

cilk_for (int j = 0; j < length; ++j)
    res2[j] = vfun_add_one(b[j]);

res3[:] = vfun_add_one(c[:]);
```

For more details on Cilk Plus and elemental functions please see the Intel(R) C++ Compiler XE 12.0 User and Reference Guide, and the article titled "[Elemental functions: Writing data parallel code in C/C++ using Intel® Cilk Plus](#)".

Intel® Threading Building Blocks (Intel® TBB)

Intel TBB is a library that offers a rich methodology to express parallelism in C++ programs and take advantage of multicore processor performance. It represents a higher-level, task-based parallelism that abstracts platform details and threading mechanism for performance and scalability while fitting smoothly into the object-oriented and generic framework of C++. Intel TBB uses a runtime-based programming model and provides developers with generic parallel algorithms based on a template library similar to the standard template library (STL).

Example :

```
#include "tbb/ParallelFor.h"
#include "tbb/BlockedRange2D.h"

void solve()
{
    parallel_for (blocked_range<size_t>(0, size, 1), [] (const
blocked_range<int> &r)
    {
        for (int i = r.begin(); i != r.end(); ++i)
            setQueen(new int[size], 0, (int)i);
    }
}
```

The Intel TBB task scheduler performs load balancing automatically, relieving the developer from the responsibility to perform that potentially complex task. By breaking programs into many small tasks, the Intel TBB scheduler assigns tasks to threads in a way that spreads out the work evenly.

Both the Intel C++ Compiler and Intel TBB support the new [C++0x lambda functions](#), which make STL and Intel TBB algorithms much easier to use. In order to use Intel's implementation of lambda expressions, one must compile the code with the `/Qstd=c++0x` compiler option.

Intel® Array Building Blocks (Intel® ArBB)

Intel ArBB can be defined in a number of ways. First and foremost, it is an API, backed by a library. No special preprocessor is needed to work with ArBB. It is a programming language extension - an "attached language" that requires a host language. Intel ArBB extends C++ for complex data parallelism including irregular and sparse matrices.

ArBB is best suited for compute-intensive, data parallel applications (often involving vector math). This API gives rise to a generalized data parallel programming solution. It frees application developers from dependencies on particular hardware architectures. It integrates with existing C++ development tools and allows parallel algorithms to be specified at a high level. ArBB is based on dynamic compilation that can remove modularity overhead. It translates high-level specifications of computations into efficient parallel implementations, taking advantage of both SIMD and thread-level parallelism.

Any ArBB program experiences two compilations. One is C++ compilation for binary distribution; the other is dynamic compilation for high performance execution. The second compilation is done by the ArBB Dynamic Engine. You need to copy data from the C++ space to the ArBB space in order for that data to be optimized for multiple cores. Data is kept in an isolated data space and is managed by the collection classes.

Example:

The first code segment shows a simple function that accepts vectors holding floating-point numbers. The computation executed adds corresponding elements of the first two vectors and stores each sum into the corresponding element of the third vector. A possible calling scenario for the function is shown, too.

```
void vecsum(float *a, float *b, float *c, int size)
{
    for (int i = 0; i < size; ++i) {
        c[i] = a[i] + b[i];
    }
}

...
float A[1024], float B[1024]; float C[1024];
...
vecsum(A, B, C, 1024);
```

The second code segment shows the same computation, but converted to use the ArBB API by binding the original arrays to vector names.

```
void vecsum(dense<f32> a,
            dense<f32> b,
            dense<f32> &c)
{
    c = a + b
}

...
float A[1024], float B[1024]; float C[1024];
dense<f32> va; bind(va, A, 1024);
dense<f32> vb; bind(vb, B, 1024);
```

```

dense<f32> vc; bind(vc, C, 1024);

. . .
call (vecsum) (va, vb, vc);

```

Win32* Threading API and Pthreads*

In some cases, developers prefer the flexibility of a native threading API. The main advantage of this approach is that the user has more control and power over threading than with the threading abstractions discussed so far in this article.

At the same time, however, the amount of code required to implement a given solution is higher, as the programmer must implement all the tedious thread implementation tasks, such as creation, scheduling, synchronization, local storage, load balancing, and destruction, which in the other cases are handled by the runtime system. Moreover, the number of cores available, which influences the correct number of threads to be created, must be determined. That can be a complex undertaking, particularly for platform-independent solutions.

Example:

```

void run_threaded_loop (int num_thr, size_t size, int _queens[])
{
    HANDLE* threads = new HANDLE[num_thr];
    thr_params* params = new thr_params[num_thr];

    for (int i = 0; i < num_thr; ++i)
    {
        // Give each thread equal number of rows
        params[i].start = i * (size / num_thr);
        params[i].end = params[i].start + (size / num_thr);
        params[i].queens = _queens;
        // Pass argument-pointer to a different
        // memory for each thread's parameter to avoid data races
        threads[i] = CreateThread (NULL, 0, run_solve,
            static_cast<void *> (&params[i]), 0, NULL);
    }

    // Join threads: wait until all threads are done
    WaitForMultipleObjects (num_thr, threads, true, INFINITE);

    // Free memory
    delete[] params;
    delete[] threads;
}

```

Threaded Libraries

Another way to add parallelism to an application is to use threaded libraries such as Intel® Math Kernel Library (Intel® MKL, not part of Intel Parallel Composer) and Intel® Performance Primitives (Intel® IPP). Intel MKL offers highly optimized threaded math routines for maximum performance, using OpenMP for threading. To take advantage of threaded Intel MKL functions, simply set the `OMP_NUM_THREADS` environment variable to a value greater than one. Intel MKL has internal thresholds to determine whether to perform a computation in parallel or serial, or

the programmer can manually set thresholds using the OpenMP API, specifically the `omp_set_num_threads` function. The online technical notes have some additional information about MKL parallelism ([MKL 9.0 for Windows*](#), [Intel® MKL 10.0 threading](#)).

Intel IPP is an extensive library of multicore-ready, highly optimized software functions particularly well suited to multimedia data processing and communications applications. Intel IPP uses OpenMP for threading, as well. The online technical notes provide more information about [IPP threading and OpenMP support](#).

The Intel C++ Compiler also provides an implementation of the STL `valarray` using Intel IPP for data-parallel performance of math and transcendental operations. The [C++ valarray template class](#) consists of array operations that support high-performance computing. These operations are designed to take advantage of low-level hardware features such as vectorization. The Intel implementation of `valarray` provides Intel IPP-optimized versions of several `valarray` operations through an optimized replacement `valarray` header file without requiring any source code changes. To optimize `valarray` loops with Intel Optimized Performance header files, use the `/Quse-intel-optimized-headers` compiler option.

Auto-Parallelization

Auto-parallelization is a feature of the Intel C++ Compiler. In auto-parallelization mode, the compiler automatically detects parallelism inherent in the program. The auto-parallelizer analyzes the dataflow of the loops in the application source code and generates multithreaded code for those loops which can safely and efficiently be executed in parallel. If data dependencies are present, loop restructuring may be needed for the loops to be auto-parallelized.

In auto-parallelization mode, all parallelization decisions are made by the compiler, and the developer does not have any control over which loops are to be parallelized. Auto-parallelization can be combined with OpenMP to achieve higher performance. When combining OpenMP and auto-parallelization, OpenMP will be used to parallelize loops containing OpenMP directives and auto-parallelization will be used to parallelize non-OpenMP loops. Auto-parallelization is enabled with the `/Qparallel` compiler option.

Example:

```
#define N 10000
float a[N], b[N], c[N];

void f1() {
    for (int i = 1; i < N; i++)
        c[i] = a[i] + b[i];
}

> icl /c /Qparallel par1.cpp
par1.cpp(5): (col. 4) remark: LOOP WAS AUTO-PARALLELIZED.
```

By default, the auto-parallelizer reports which loops were successfully auto-parallelized. Using the `/Qpar-report[n]` option, where `n` is a number between 0 and 3, the auto-parallelizer can report diagnostic information about auto-parallelized loops and those that did not get auto-parallelized. For example, `/Qpar-report3` tells the auto-parallelizer to report diagnostics messages for loops successfully and unsuccessfully auto-parallelized plus information about any proven or assumed dependencies inhibiting auto-parallelization. The diagnostics information helps restructure loops to be auto-parallelized.

Auto-Vectorization

Vectorization is the technique used to optimize loop performance on Intel® processors. Parallelism defined by vectorization technique is based on vector-level parallelism (VLP) made possible by the processor's [SIMD](#) hardware. The auto-vectorizer in the Intel C++ Compiler automatically detects low-level operations in the program that can be performed in parallel and then converts the sequential code to process 1-, 2-, 4-, 8-, or up to 16-byte data elements in one operation with extensions up to 32- and 64-byte in the future processors. Loops need to be independent for the compiler to auto vectorize them. Auto-vectorization can be used in conjunction with the other thread-level parallelization techniques such as auto-parallelization and OpenMP discussed earlier. Most floating-point applications and some integer applications can benefit from vectorization. The default vectorization level is `/arch:SSE2` which generates code for Intel® Streaming SIMD Extensions 2 (Intel® SSE2). To enable auto-vectorization for other than the default target, use the `/arch` (e.g., `/arch:SSE4.1`) or `/Qx` (e.g., `/QxSSE4.2`, `QxHost`) compiler options.

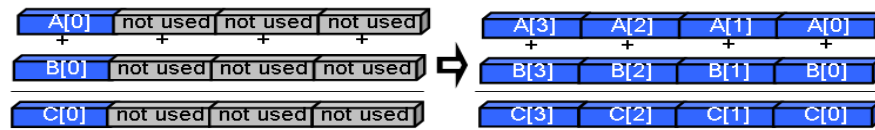
The figure below shows the serial execution of the loop iterations on the left without vectorization, where the lower parts of the SIMD registers are not utilized. The vectorized version on the right shows four elements of the A and B arrays added in parallel for each iteration of the loop, utilizing the full width of the SIMD registers.

Data parallelism

- **Generates SSE instructions**
- **Operate at once on 2 double, 4 float and int etc.**

Example:

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



Note: A, B and C are 128 bit SIMD registers

Figure 1. Loop iterations with and without vectorization.

Example:

```
#define N 10000  
float a[N], b[N], c[N];  
  
void f1() {  
    for (int i = 1; i < N; i++)  
        c[i] = a[i] + b[i];  
}  
  
> icl /c /QxSSE4.2 par1.cpp  
par1.cpp(5): (col. 4) remark: LOOP WAS VECTORIZED.
```

By default, the vectorizer reports which loops got vectorized. Using the `/Qvec-report[n]` option, where `n` is a number between 0 and 5, the vectorizer can report diagnostic information about vectorized and non-vectorized loops. For example, the `/Qvec-report5` option tells the vectorizer to report on non-vectorized loops and the reason why they were not vectorized. The diagnostics information helps restructure the loops to be vectorized.

Advice/Usage Guidelines

Tradeoffs Between Different Methods

Various parallelism methods can be categorized in terms of abstraction, control, and simplicity. Intel® TBB, Intel® ArBB, and the API models do not require specific compiler support, but Intel® Cilk™ Plus, and OpenMP do. The use of Intel Cilk Plus, and OpenMP require the use of a compiler that recognizes Cilk Plus keywords and array syntax, and the OpenMP directives. The API-based models require the programmer to manually map concurrent tasks to threads. There is no explicit parent-child relationship between the threads; all threads are peers. These models give the programmer control over all low-level aspects of thread creation, management, and synchronization. This flexibility is the key advantage of library-based threading methods. The tradeoff is that to obtain this flexibility, significant code modifications and a lot more coding are required. Effort spent on performance tuning will often not scale up or down to different core counts or operating system versions. Concurrent tasks must be encapsulated in functions that can be mapped to threads. The other drawback is that most threading APIs use arcane calling conventions and only accept one argument. Thus, it is often necessary to modify function prototypes and data structures that may break the abstraction of the program design, which fits better in a C approach than an Object-oriented C++ one.

As a compiler-based threading method, OpenMP provides a high-level interface to the underlying thread libraries. With OpenMP, the programmer uses OpenMP directives to describe parallelism to the compiler. This approach removes much of the complexity of explicit threading methods, because the compiler handles the details. Due to the incremental approach to parallelism, where the serial structure of the application stays intact, there are no significant source code modifications necessary. A non-OpenMP compiler simply ignores the OpenMP directives, leaving the underlying serial code intact.

With OpenMP, however, much of the fine control over threads is lost. Among other things, OpenMP does not give the programmer a way to set thread priorities or perform event-based or inter-process synchronization. OpenMP is a fork-join threading model with an explicit master-worker relationship among threads. These characteristics narrow the range of problems for which OpenMP is suited. In general, OpenMP is best suited to expressing data parallelism, while explicit threading API methods are best suited for functional decomposition. OpenMP is well known for its support for loop structures and C code, but it offers nothing specific for C++. OpenMP version 3.0 supports tasking, which extends OpenMP by adding support for irregular constructs such as `while` loops and recursive structures. Nevertheless, OpenMP remains reminiscent of plain C and FORTRAN programming, with minimal support for C++.

Intel ArBB provides a generalized vector parallel programming solution that frees application developers from dependencies on particular low-level parallelism mechanisms or hardware architectures. Intel ArBB uses C++ language extensions for compatibility with all standard compilers and IDEs and is not tied to a particular compiler.

Intel TBB supports generic scalable parallel programming using standard C++ code like the STL. It does not require special languages or compilers. If one needs a flexible and high-level parallelization approach that fits nicely in an abstract and even generic object-oriented approach, Intel TBB is an excellent choice. Intel TBB uses templates for common parallel iteration patterns and supports scalable data-parallel programming with nested parallelism. In comparison to the API approach, one specifies tasks rather than threads, and the library maps tasks onto threads in

an efficient way using the Intel TBB runtime. The Intel TBB scheduler favors a single, automatic divide-and-conquer approach to scheduling. It implements task stealing, which moves tasks from loaded cores to idle ones. In comparison to OpenMP, the generic approach implemented in Intel TBB allows developer-defined parallelism structures that are not limited to built-in types.

Intel Cilk Plus supports both data and task-parallelism in both C and C++ language. With its three simple keywords that provide simple fork-join parallelism, Cilk Plus is the easiest way to introduce parallelism to an existing serial program. It provides the lowest overhead among the parallel models discussed here to invoke parallel threads. The cilk keywords can be ignored by a non-Cilk compiler simply by using the preprocessor to replace the keywords with their serial equivalents (A header file is provided for this purpose.) However, the array notation cannot be so easily elided. Unlike OpenMP that does not compose well with itself (e.g. nested OpenMP) and other threading models, Intel Cilk Plus composes well with Intel TBB and Intel ArBB without causing thread over subscription. This allows a programmer to use Cilk Plus for the majority of the code, and use Intel TBB to implement parallelism where other parallel constructs and parallel data structures are needed such as scoped lock, parallel hash, etc.

The following table compares the different threading techniques available in Intel Parallel Composer:

Method	Description	Benefits	Caveats
Explicit Threading APIs	Low-level APIs such as the Win32* Threading API and Pthreads* for low-level multi-threaded programming	<ul style="list-style-type: none"> • Maximum control and flexibility • Does not need special compiler support 	<ul style="list-style-type: none"> • Relatively complex code to write, debug, and maintain; very time-consuming • All thread management and synchronization done by the programmer
OpenMP* (Enabled by /Qopenmp compiler option)	A specification defined by OpenMP.org to support shared-memory parallel programming in C/C++ and Fortran through the use of APIs and compiler directives	<ul style="list-style-type: none"> • Potential for large performance gain with relatively little effort • Good for rapid prototyping • Can be used for C/C++, and Fortran • Allows incremental parallelism using compiler directives • User control over what code to parallelize • Single-source solution for multiple platforms • Same code base for both serial and parallel version 	Not much user control over threads such as setting thread priorities or performing event-based or inter-process synchronization

Intel® Cilk™ Plus	New Keywords for C and C++ (<code>cilk spawn</code> , <code>cilk sync</code> , <code>cilk for</code>), reducers to avoid race conditions, and array notations to take advantage of vectorization.	<ul style="list-style-type: none"> • Clean syntax that preserves serial organization and semantics of the original program. • Multicore and SIMD in a single package. • Composable, easy to reason about. 	<ul style="list-style-type: none"> • Requires compiler support • No support for Fortran • No fine-grain control over threading
Intel® Threading Building Blocks	Intel's C++ runtime library that simplifies threading for performance by providing parallel algorithms and concurrent data structures that eliminate tedious threading implementation work	<ul style="list-style-type: none"> • Does not need special compiler support • Uses standard C++ code like STL • Automatic thread creation, management, and scheduling • Allows expressing parallelism in terms of tasks rather than threads 	<ul style="list-style-type: none"> • Mostly suited to C++ programs • No support for Fortran
Intel(R) ArBB	An Intel® standard library interface and runtime for compute-intensive, data parallel applications that uses C++ language extensions for compatibility with all standard compilers and IDEs.	<ul style="list-style-type: none"> • Simple syntax invites users to focus on high-level algorithms. • Syntax does not allow aliases, allowing aggressive optimization by the runtime. • Separate memory space for Intel ArBB and C/C++ objects. Intel ArBB objects can only be operated on by Intel ArBB functions. • Developers do not use thread, locks or other lower-level constructs and can avoid the associated complexity. 	<ul style="list-style-type: none"> • Not a substitute for hand-tuned code. • Giving more control over memory, threading, and vectorization to ArBB runtime compiler. • JIT-compile process creates scenario where the first run incurs high amounts of overhead. • Have to convert all your data types to ArBB types and use the ArBB function interface.
Auto-Parallelization (Enabled by <code>/Qparallel</code> compiler option)	A feature of the Intel® C++ Compiler to automatically parallelize loops with no loop-carried dependency in a program	<ul style="list-style-type: none"> • Compiler automatically generates multi-threaded code for parallelizable loops • Can be used together with other threading techniques 	Works on loops that compiler can statically prove are parallelizable through data-dependency and aliasing analysis

Auto-Vectorization (Enabled by /arch: and /Qx options)	Technique used to optimize loop performance through vector-level parallelism on Intel® processors by converting sequential instructions to SIMD instructions that can operate on multiple data elements at once	<ul style="list-style-type: none"> • Automatic vector level parallelism done by the compiler • Can be used together with other threading techniques 	Resulting code may not run on all processors if processor-specific options are used
--	---	---	---

The article, "[Solve the N-Queens problem in parallel](#)," provides hands-on training about applying each of the parallelization techniques discussed in this document to implement a parallel solution to the N-Queens problem, which is a more general version of the [Eight Queens Puzzle](#). Additional examples are provided in the "Samples" folder under the Intel® C++ Compiler installation folder.

Additional Resources

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

[General information on Intel Compilers, documentation, White Papers, Knowledge Base](#)

[The Software Optimization Cookbook \(2nd Edition\) High performance Recipes for the Intel Architecture](#)

[Intel Software Network Forums](#)

[Additional information on OpenMP, including the complete specification and list of directives](#)

[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.

[Intel® C++ Compiler Optimization Guide](#)

[Quick Reference Guide to optimization with the Intel Compilers](#)

Optimize Data Structures and Memory Access Patterns to Improve Data Locality

Abstract

Cache is one of the most important resources of modern CPUs: it's a smaller and faster part of the memory sub-system where copies of the most frequently used memory locations are stored. When data required by an instruction reside in the cache the instruction will be executed immediately. Otherwise, execution of instructions might get suspended until the required data is fetched from the memory. Since copying data from the memory is a long latency operation, we'd like to minimize cache misses by designing algorithms and data structures to exploit data locality.

This article discusses symptoms of poor data locality, techniques to detect related performance bottlenecks, and possible optimizations to cure the problem.

Background

A classical example that demonstrates the problem of poor data locality and its impact is a naïve implementation of matrix multiplication algorithm. Let's start with serial version of the algorithm that multiplies square matrices A and B and stores result in matrix C according to the following formula: $C_{ij} = \sum A_{ik} B_{kj}$; each matrix is stored in an array in row-major order and the size of each matrix is quite big ($N \times N$):

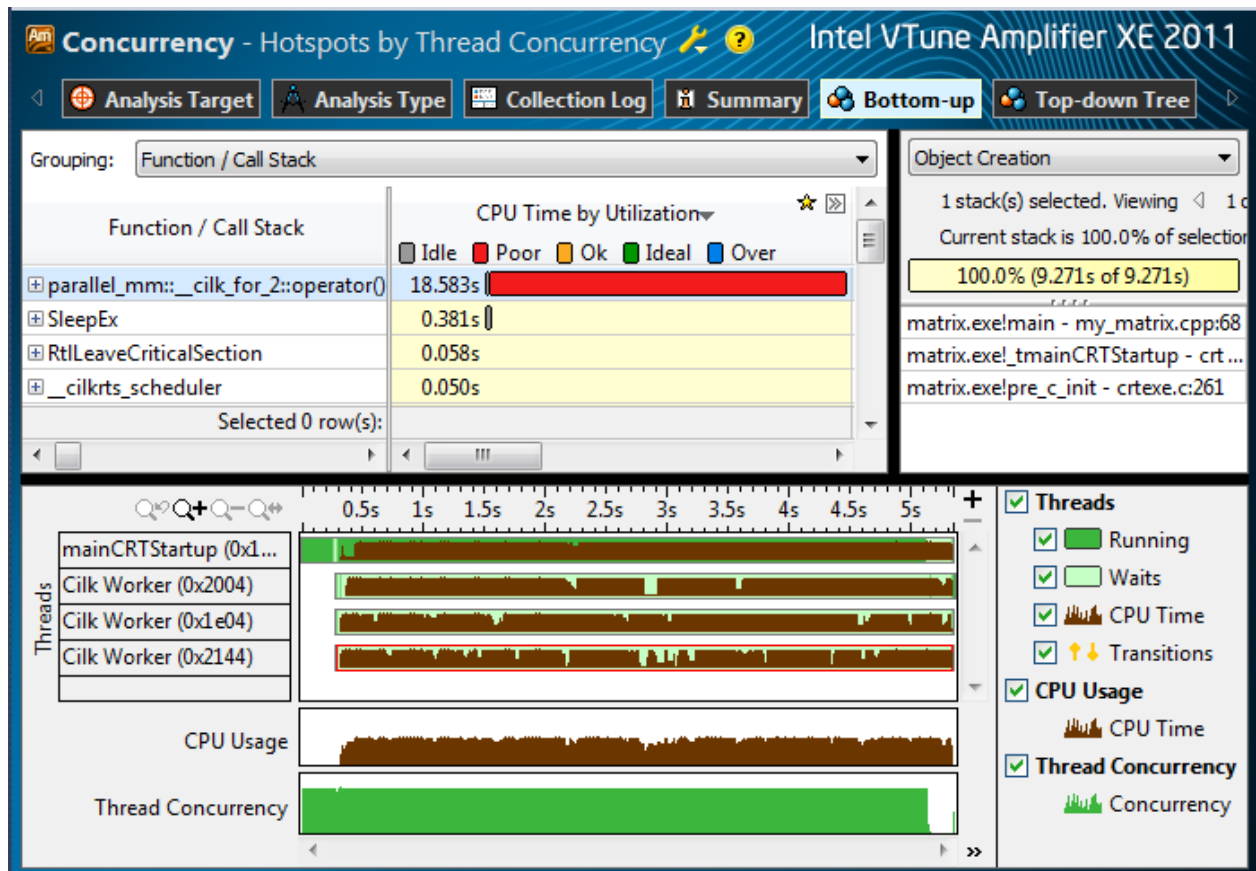
```
void serial_mm( PType& A, PType& B, PType& C )
{
    for( size_t i = 0; i < N; ++i ) {
        for( size_t j = 0; j < N; ++j )
            for( size_t k = 0; k < N; k++ )
                C[i*N+j] += A[i*N+k] * B[j+k*N];
    }
}
```

This is a trivial example of the compute-intensive algorithm which shares common characteristics with many other algorithms: 1) it's implemented using nested for-loops; and 2) it manipulates large chunk of data. Another, less obvious, characteristic is that the iterations of the outer loop are independent and it seems to be a perfect candidate for parallelization. The following code shows a parallel version of the algorithm implemented with Intel® Cilk™ Plus (we just replaced key-word *for* with Intel® Cilk Plus™ *cilk_for* and used the Intel® Compiler that supports Intel Cilk Plus language extension):

```
void parallel_mm( PType& A, PType& B, PType& C, bool in_order )
{
    cilk_for( size_t i = 0; i < N; ++i ){
        for( size_t j = 0; j < N; ++j )
            for( size_t k = 0; k < N; k++ )
                C[i*N+j] += A[i*N+k] * B[j+k*N];
    }
}
```

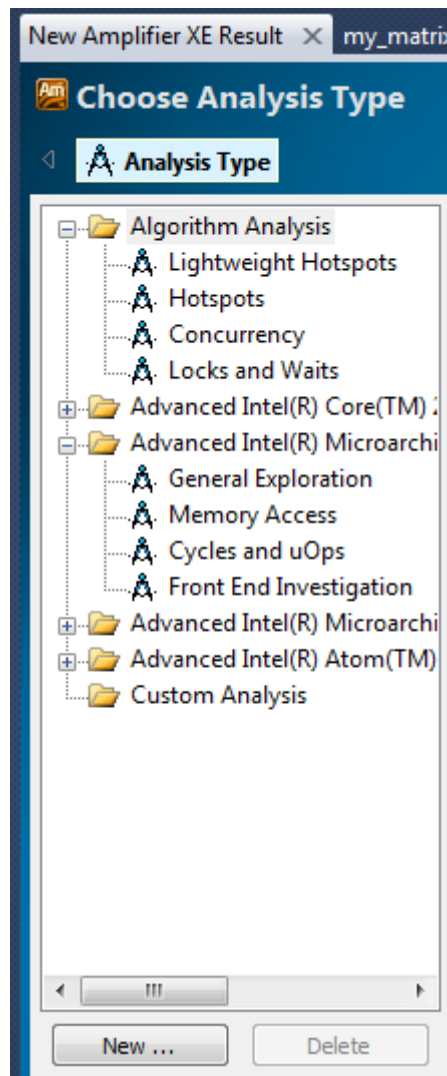
Comparing performance of the parallel implementation vs. serial for big matrices, we discover that parallel version actually runs slower.

This is one of the examples where high-level analysis is not helpful in root causing the problem. Indeed, if we run Intel® VTune™ Amplifier XE Concurrency analysis we see that there is a problem: a poor concurrency level while running a hot-spot function (parallel_mm), but it doesn't tell us why:



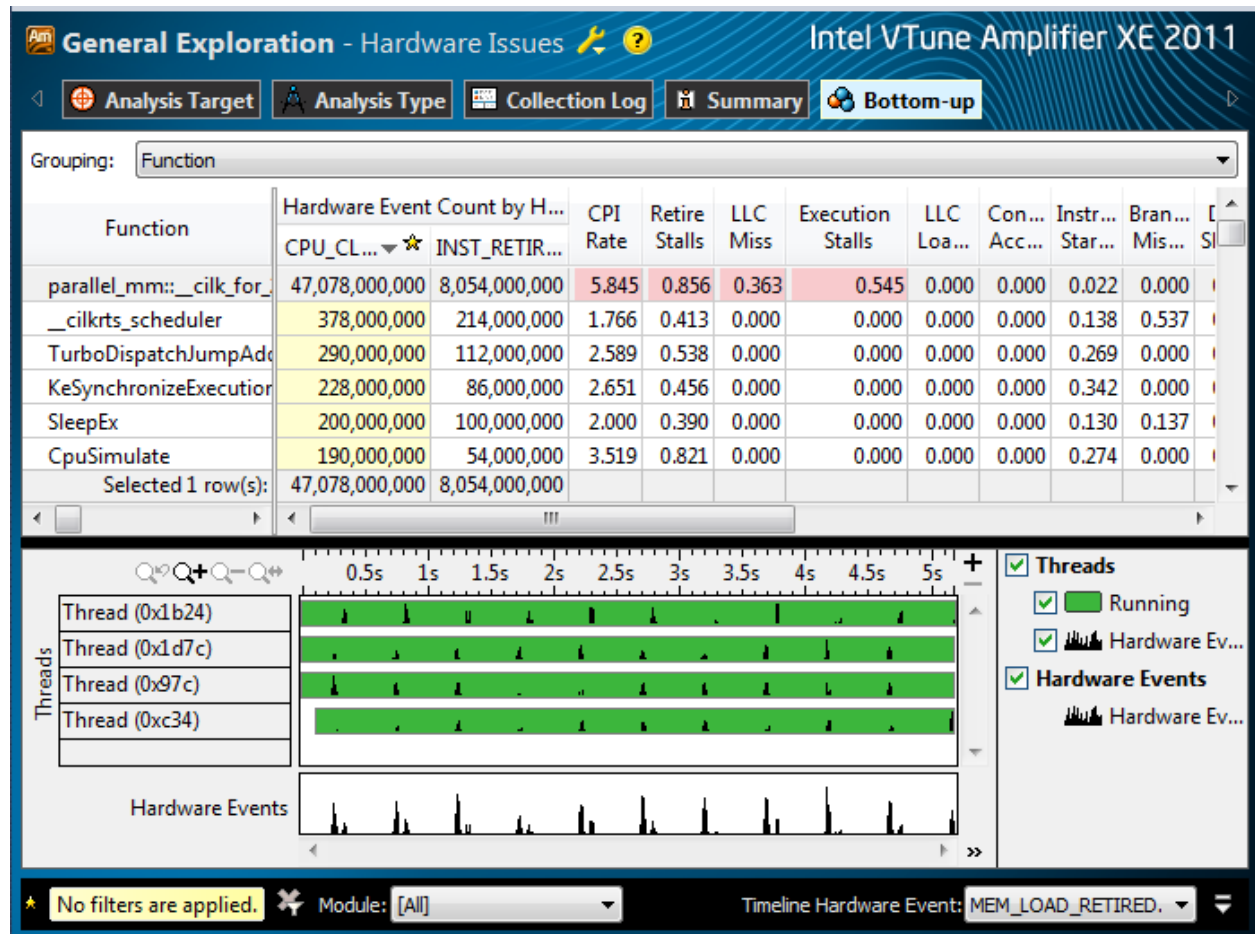
We could get a similar result running Intel® Parallel Amplifier's Concurrency analysis. However, using Intel VTune Amplifier XE, extreme edition of Intel Parallel Amplifier, adds one more dimension to the presentation of the performance statistics: the timeline view. The timeline view shows the application's worker threads and their behavior over time. In case of the concurrency analysis applied to the matrix multiplication program, Intel® VTune™ Amplifier XE detected three worker threads created by Intel® Cilk™ Plus' run-time system in addition to the main thread (marked as "Cilk Worker" on the timeline view), and distribution of the CPU Time per worker thread over time (shown in brown color on the timeline). Based on the data collected by Concurrency analysis we conclude that while CPU utilization by worker threads is very low, CPU Time is very high throughout the execution of the algorithm.

The next step in attempt to find the cause of the poor concurrency is to run one of the low-level profilers supported by Intel VTune Amplifier XE:



Intel VTune Amplifier XE supports several low-level analysis types that are based on Event-based Sampling (EBS): Lightweight Hotspots and a few analysis types grouped under the “Advanced” category. The purpose of EBS analysis is to collect hardware events by sampling special registers called PMUs (Performance Monitoring Units) which could be programmed to increment their values when a certain hardware event occurs (e.g. branch misprediction, cache miss etc). CPU can generate many different events and looking at one of them or a group might not always be helpful because often it is unclear whether the numbers that the profiler shows you are really the ones that are responsible for bad performance. Often, the right combination of events and a certain formula have to be used in order to confirm whether or not there is a particular performance issue in the application. If you run low-level profiler for the first time or you are not sure what type of problem you should be looking for, it is recommended that you start with “General Exploration.” “General Exploration” is a pre-configured analysis that collects multiple hardware events and uses a number of formulas which might help to understand what type of problem the application has: the tool collects hardware events, shows data that are the result of some formula, and highlights numbers that might indicate a performance problem. All events and formulas are described in the Intel VTune Amplifier XE Help in great detail; a shorter description can be seen on tooltips.

Running "General Exploration" for matrix multiplication program reveals several performance bottlenecks (cells marked with pink):

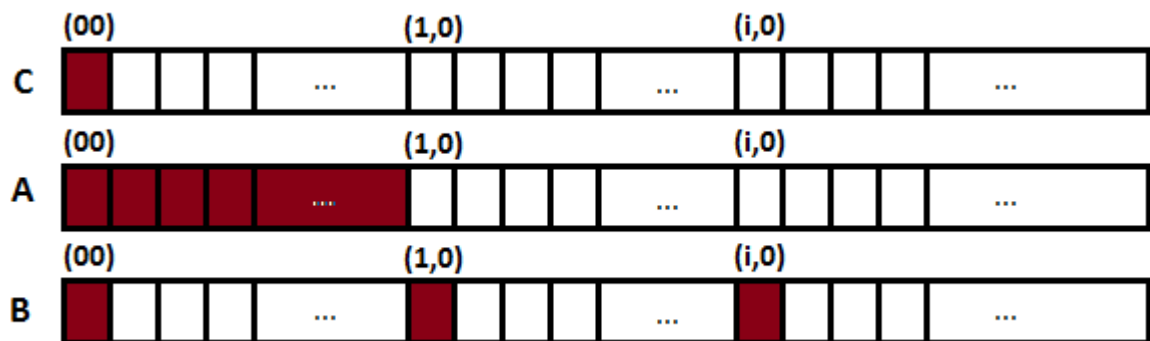


1. **CPI Rate:** Shows how many CPU cycles were spent per instruction on average. Ideally, we'd like this number to be less or equal to 1 but in our case it's greater than 5.
2. **Retire Stalls:** Shows a ratio of the number of cycles when no micro-operations are retired to all cycles. In the absence of long latency operations and dependency chains, this number should be equal or greater than 1, in our case it's ~0.8.
3. **LLC Miss:** Shows a ratio of cycles with outstanding LLC (Last-level Cache) misses to all cycles. Any memory requests that miss LLC must be serviced by local or remote DRAM, with significant latency. Ideally, this number should be very close to 0.
4. **Execution Stalls:** Shows a ratio of cycles with no micro-operations executed to all cycles. Execution Stalls are caused by waiting on critical computational resource.

A conclusion that can be made based on the data shown by Intel VTune Amplifier XE is that CPU spends time waiting for data to be fetched from a local or remote DRAM due to the high rate of LLC misses. When we double-click on the function, Intel VTune Amplifier XE will show a function source view and distribution of the hardware events per source line. For our example, most of the samples were generated by the inner-most for-loop:

General Exploration - Hardware Issues				
<div> Analysis Target Analysis Type Collection Log Summary Bottom-up my_matr... </div>				
<div> Source Assembly Call Graph Data Explorer Performance Monitor Help </div>				
Line	Source	CPU_CLK_... THREAD by Package	INST_RETIRED. ANY by Package	CACH- L1 Pa
37	}			
38				
39	void parallel_mm(PType& A, PType& B, PType& C)			
40	{			
41	cilk_for(size_t i = 0; i < N; ++i) {			
42	for(size_t j = 0; j < N; ++j)	2,000,000		
43	for(size_t k = 0; k < N; k++)	43,182,000,000	7,878,000,000	
44	C[i*N+j] += A[i*N+k] * B[j+k*N];	3,894,000,000	176,000,000	
45	}			
46	}			

Let's examine the data access pattern implemented in parallel_mm function for $i = 0$ and $j = 0$:



Each iteration of the inner-most loop touches an element of matrix A that is close to the element touched on the previous iteration and such an access pattern exploits efficient data locality. Access pattern for matrix B, however, is not efficient because on each iteration the program needs to access j -th element of k -th column where k is an inner loop counter. Such an inefficient data access pattern is the reason of the program's poor performance because it leads to the high rate of cache misses.

Let's consider a more efficient implementation of the algorithm which requires only a small change in the code which dramatically improves the data access pattern:

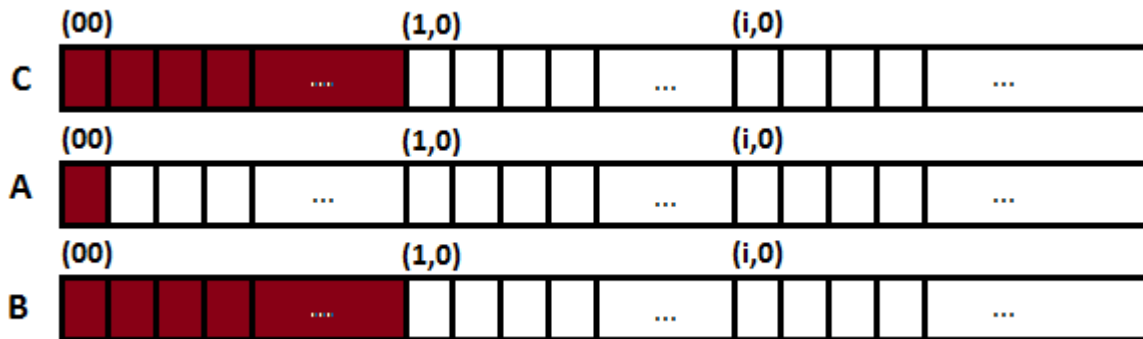
```
void parallel_mm( PType& A, PType& B, PType& C )
{
    // Note the changed loop order
```

```

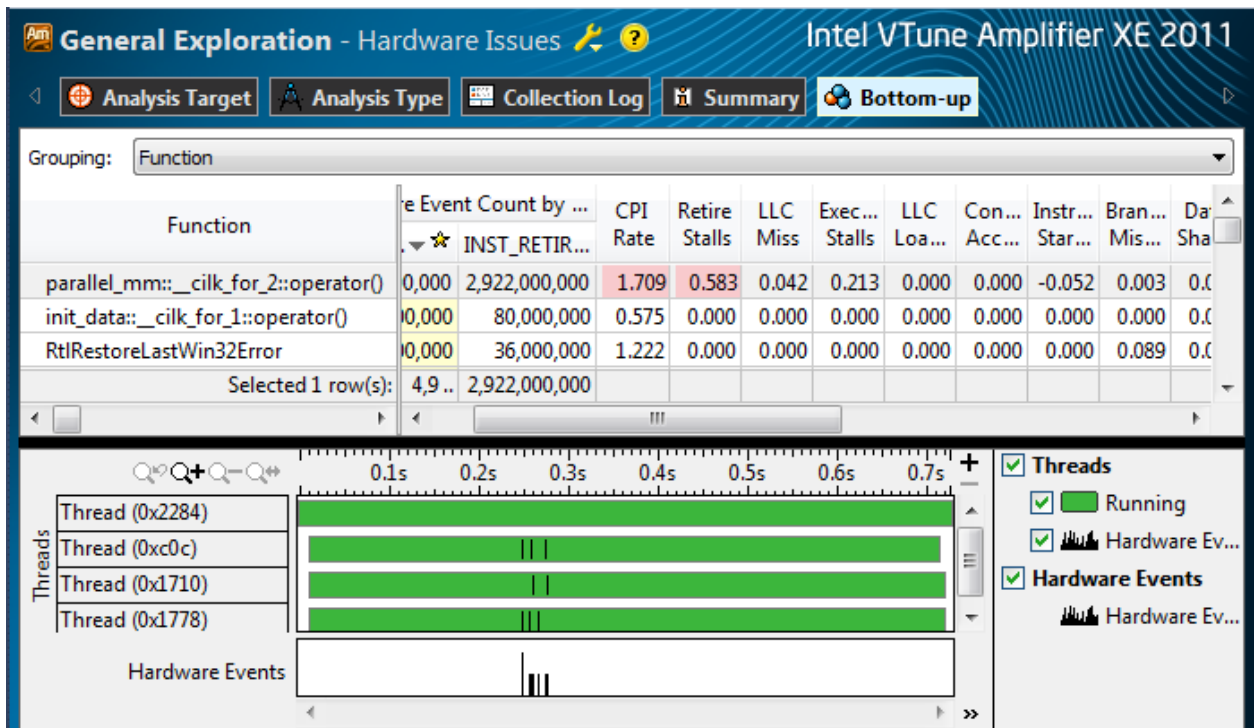
    cilk_for( size_t i = 0; i < N; ++i ) {
        for( size_t k = 0; k < N; ++k )
            for( size_t j = 0; j < N; j++ )
                C[i*N+j] += A[i*N+k] * B[j+k*N];
    }
}

```

Exchanging two inner loops leads to an improved data locality (i=0, k=0):



With this new implementation, for each i (counter of the outer loop) we calculate the entire row of matrix C and touch only consecutive elements of matrices A and B. This change greatly improves performance by increasing CPI and minimizing LLC misses:



Advice

Intel® VTune Amplifier XE has many useful features that can help find performance bottlenecks in your program. While high-level analysis types can often help with algorithmic optimizations, low-level analysis types focus on hardware behavior during the application execution. Modern CPUs have become very complicated and it might take an expert to perform low-level performance analysis. However, Intel VTune Amplifier XE's philosophy is to simplify low-level performance analysis and make it more accessible to non-expert users. When high-level analysis types cannot help to figure out the problem, try running Intel VTune Amplifier XE General Exploration which might give you a hint as to what to do next and localize the problem.

In the previous section we discussed only one example of the data locality problem and the change of the algorithm helped us improve data locality and performance. However, poor data locality can be caused by poorly designed data structures and it might be impossible to improve data locality without changing a layout of data. This second type of problems related to a data locality will still have the same symptoms that can be detected by Intel VTune Amplifier XE: low concurrency levels, poor CPI, and a high rate of cache misses

Design your data structures and algorithms with data locality in mind to achieve good performance and scalability. Both serial and parallel applications will benefit from exploiting a good data locality.

Usage Guidelines

Performance analysis is an iterative process. When using one of the Intel VTune Amplifier XE analysis types, re-run it after introducing changes to your code. Intel VTune Amplifier XE provides an automatic comparison feature so that you can compare two profiles for two different runs and track your progress that way.

In this article we discussed "General Exploration" analysis. Its purpose is to help you get started with the low-level performance analysis. There are other, more specific, analysis types available ("Memory Access," "Cycles and uOps," etc.) which can be run as a next step or when you know what type of performance bottleneck you are investigating, or if you need more specific details.

Additional Resources

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

[Intel\(R\) VTune\(TM\) Amplifier XE product page](#)