Automatic Parallelization with Intel® Compilers

<keywords: threading, auto-parallelization, data dependences, programming tools, compiler>

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.

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

Background

The 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 most 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 (!

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:

```c
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:

```c
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 autoparallelization. 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


Efficient Exploitation of Parallelism on Pentium® III and Pentium® 4 Processor-Based Systems

An Overview of the Parallelization Implementation Methods in Intel® Parallel Composer