Intel OpenMP C++/Fortran Compiler for Hyper-Threading Technology: Implementation and Performance

Xinmin Tian, Software Solutions Group, Intel Corporation
Aart Bik, Software Solutions Group, Intel Corporation
Milind Girkar, Software Solutions Group, Intel Corporation
Paul Grey, Software Solutions Group, Intel Corporation
Hideki Saito, Software Solutions Group, Intel Corporation
Ernesto Su, Software Solutions Group, Intel Corporation

Index words: Hyper-Threading Technology, OpenMP, Optimization, Parallelization, Vectorization

ABSTRACT

In the never-ending quest for higher performance, CPUs become faster and faster. Processor resources, however, are generally underutilized by many applications. Intel's Hyper-Threading Technology is developed to resolve this issue. This new technology allows a single processor to manage data as if it were two processors by executing data instructions from different threads in parallel rather than serially. Processors enabled with Hyper-Threading Technology can greatly improve the performance of applications with a high degree of parallelism. However, the potential gain is only obtained if an application is multithreaded, by either manual, automatic, or semi-automatic parallelization techniques. This paper presents the compiler techniques of OpenMP pragma- and directive-guided parallelization developed for the high-performance Intel C++/Fortran compiler. We also present a performance evaluation of a set of benchmarks and applications.

INTRODUCTION

Intel processors have a rich set of performance-enabling features such as the Streaming-SIMD-Extensions (SSE and SSE2) in the IA-32 architecture [11], large register files, predication, and control and data speculation in the Itanium-based architecture [8]. These features allow the compiler to exploit parallelism at various levels. Intel's newest Hyper-Threading Technology [14], a simultaneous multithreading design, allows one physical processor to manage data as if it were two logical processors by handling data instructions in parallel rather than serially. The Hyper-Threading Technology-enabled processors can significantly increase the performance of application programs with a high degree of parallelism. These potential performance gains are only obtained, however, if an application is efficiently multithreaded, either manually or by automatic or semi-automatic parallelization techniques. The Intel C++/Fortran high-performance compiler supports several such techniques. One of those techniques, automatic loop parallelization, was presented in [3]. In addition to automatic loop level parallelization, Intel compilers support OpenMP directive- and pragma-guided parallelization as well, which significantly increase the domain of various applications amenable to effective parallelism. For example, users can use OpenMP parallel sections to develop an application where *section-1* calls an integer-intensive routine and where *section-2* calls a floating-point intensive routine. Higher performance is obtained by scheduling *section-1* and *section-2* onto two different logical processors that share the same physical processor to fully utilize processor resources based on the Hyper-Threading Technology. The OpenMP standard API [12, 13] supports a multi-platform, shared-memory, parallel programming paradigm in C++/C/Fortran95 on all Intel architectures and popular operating systems such as

Intel is a registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

Windows NT , Linux*, and Unix*. OpenMP directives and pragmas have emerged as the de facto standard of expressing parallelism in various applications as they substantially simplify the notoriously complex task of writing multithreaded programs.

The Intel compilers support the OpenMP pragmas and directives in the languages C++/C/Fortran95, on Windows* and Linux platforms and on IA-32 and IPF architectures. The Intel OpenMP implementation in the compiler strives to (i) generate multithreaded code which gains a speed-up due to Hyper-Threading Technology over optimized uniprocessor code, (ii) integrate parallelization tightly with advanced scalar and loop optimizations such as intra-register vectorization [4] and memory optimizations [1, 10] to achieve better cache locality and efficiently exploit multi-level parallelism, and (iii) minimize the overhead of data-sharing among threads.

This paper focuses on the design and implementation of OpenMP pragma- and directive-guided parallelization in the Intel® C++/Fortran compilers. We also present performance results of a number of applications (Micro-benchmark, Image processing library functions, OpenMP benchmarks from [2]) that exhibit performance gains due to Hyper-Threading Technology when such programs are multithreaded through the OpenMP directives or pragmas and compiled with Intel C++/Fortran compilers.

The remainder of this paper is organized as follows. We first give a high-level overview of the architecture of the Intel C++/Fortran compiler with OpenMP support. We then present the Multi-Entry Threading (MET) technique that is the key technique developed for multithreaded code generation in the Intel compilers. We go on to describe the local static data-sharing and privatization methods for minimizing overhead of data sharing among threads. We briefly explain how OpenMP parallelization interacts with advanced optimizations such as constant propagation, interprocedural optimization, and partial redundancy elimination. We also briefly describe how multi-level parallelism is exploited by combining parallelization with intra-register vectorization to take advantage of the Intel Pentium 4 processor SIMD-Streaming-Extensions (SSE and SSE2). Finally, we show the performance results of several OpenMP benchmarks and applications when such programs are multithreaded by the Intel OpenMP C++/Fortran compilers.

HIGH-LEVEL COMPILER OVERVIEW

A high-level overview of the Intel® OpenMP C++/Fortran compiler is shown in Figure 1. The compiler incorporates many well-known and advanced optimization techniques that are designed and extended to fully leverage Intel processor features for higher performance. The Intel compiler has a common intermediate representation for C++, C and Fortran95 languages, so that the OpenMP directive- or pragma-guided parallelization and a majority of optimization techniques are applicable through a single high-level code transformation, irrespective of the source language. Throughout the rest of this paper, we refer to Intel OpenMP C++ and Fortran compilers for IA-32 and Itanium processor family architectures collectively as "the Intel compiler."

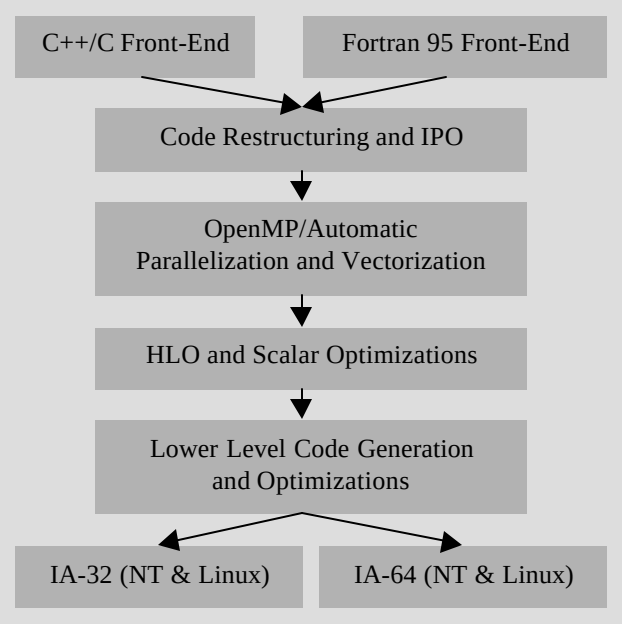

Figure 1: Compiler architecture overview

The code transformations and optimizations in the Intel compiler can be classified into (i) code restructuring and interprocedural optimizations (IPO); (ii) OpenMP-based and automatic parallelization and vectorization; (iii) high-level optimizations (HLO) and scalar optimizations including memory optimizations such as loop control and data transformations, partial redundancy elimination (PRE) [7], and partial dead store elimination (PDSE); and (iv) low-level machine code generation and optimizations such as register allocation and instruction scheduling.

Parallelization [3, 4, 10] guided by OpenMP directives or pragmas or derived by automatic data dependency and control-flow analysis is a high-level code transformation that exploits both medium- and coarse-grained parallelism for Intel processor and multiprocessor systems enabled with Hyper-Threading Technology to achieve better performance and higher throughput. The Intel compiler has a common intermediate code representation (called IL0) into which C++/C and Fortran95 programs are translated by the language front-ends. Many optimization phases in the compiler work on the IL0 representation. The IL0 has been extended to express the OpenMP directives and pragmas. Implementing the OpenMP phase at the IL0 level allows the same implementation to be used across languages (C++/C, Fortran95) and architectures (IA-32 and IPF). The Intel compiler-generated code has references to a high-level multithreaded library API; this allows the compiler OpenMP transformation phase to be independent of the underlying operating systems. This also facilitates our "one-for-all" design philosophy.

A primary goal of the Intel compiler is to have OpenMP parallelization tightly integrated with advanced compiler optimizations for generating efficient multithreaded code that gains a speed-up over optimized uniprocessor code. Therefore, an effective optimization phase ordering has been designed in the Intel compiler to make sure that all optimizations, such as IPO inlining, code restructuring; Igoto optimizations, and constant propagation, which are effectively enabled before the OpenMP parallelization, preserve legal OpenMP program semantics and necessary information for parallelization. It also ensures that all optimizations after the OpenMP parallelization, such as automatic vectorization, loop transformation, PRE, and PDSE, can effectively kick in to achieve a better cache locality and to minimize the number of computations and the number of references to memory. For example, given a double-nested OpenMP parallel loop, the parallelization methods are able to generate multithreaded code for the outer loop, while maintaining the loop structure, memory reference behavior, and symbol table information for the innermost loop. This enables subsequent intra-register vectorization of the innermost loop to fully leverage the Hyper-Threading Technology and SIMD Streaming Extension features of Intel processors. Exploiting multi-level parallelism is described later in this paper.

OpenMP parallelization in the Intel compiler includes (i) a pre-pass that transforms OpenMP parallel sections and worksharing sections into a parallel loop and worksharing loop, respectively; (ii) a work-region graph builder that builds a region hierarchical graph based on the OpenMP-aware control-flow graph; (iii) a loop analysis phase for building the loop structure that consists of loop control variable, loop lower-bound, loop upper-bound, loop pre-header, loop header, and control expression; (iv) a variable classification phase that performs analysis of shared and private variables; (v) a multithreaded code generator that generates multithreaded code at compiler intermediate code level based on Guide, a multithreaded run-time library API that is provided by the Intel KAI Software Laboratory (KSL); (vi) a privatizer that performs privatization to handle firstprivate, private, lastprivate, and reduction variables; and (vii) a post-pass that generates code to cache in thread local storage for handling threadprivate variables. There are a number of compiler techniques developed for parallelization in the Intel compiler. The following sections describe some of these techniques in detail.

MULTI-ENTRY THREADING

A well-known conventional technology, which was named outlining [5, 6], has been used by existing parallelizing compilers for generating multithreaded codes. The basic idea of outlining is to generate a separate subroutine for a parallel region or loop. All threads in a team call this routine with necessary data environment. In contrast to the outlining technology, we developed and implemented a new compiler technology called *Multi-Entry Threading* (MET). The rationale behind MET is that the compiler does not create a separate compilation unit (or routine) for a parallel region or loop. Instead, the compiler generates a threaded entry and a threaded return for a given parallel region and loop [3]. Based on this idea, we introduced three new graph nodes in the Region-based graph, built on top of the control-flow graph. These graph nodes are *T-entry* (threaded entry), *T-ret* (threaded return), and *T-region* (threaded code region). A detailed description of these graph nodes is given as follows:

> T-entry indicates the entry point of a multithreaded code region and has a list of firstprivate, lastprivate, shared and/or reduction variables for communication among the threads in a team.
>
> T-ret indicates the exit point of a multithreaded code region and guides the lower-level target machine code generator to adjust stack offset properly and give the control to the caller inside the runtime library. T-region represents a multithreaded code region that is attached inside the original user routine.

The main concept of the MET compilation model is to keep all newly generated multithreaded codes, which are captured by *T-entry*, *T-region* and *T-ret* nodes, intact or inlined within the same user-defined routine without splitting them into independent subroutines. This method provides later compiler optimizations with more

opportunities for performing optimization. Example (E1-I) is an OpenMP program sample.

Given the parallel program with OpenMP pragmas above, its region-based hierarchical graph is shown in Figure 2. As we see, the first *T-region* represents the OpenMP parallel sections and the second *T-region* represents the OpenMP parallel loop in the routine '*parfoo*.' Each *T-region* contains a *T-entry* node and a *T-ret* node. With OpenMP data attribute clauses, the variables '*w*' and '*y*' are marked as *shared* and the arrays '*x*' and '*z*' are marked as *shared* as well in the parallel sections clause. For the parallel loop, the loop control variable 'm' is marked as *private*, and the variables '*y*' and '*w*' and the array '*z*' are marked as *shared*. The *guided* scheduling type is specified for the parallel loop. The generated pseudo-multithreaded code is shown below in (E1-II). As mentioned previously, the Intel KSL Guide runtime library API has been adopted for thread creation, synchronization and scheduling.

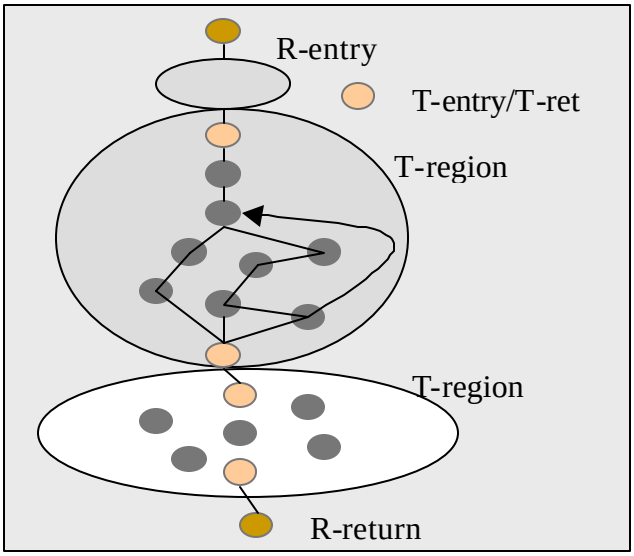

Figure 2: Region-based hierarchical graph

(E1-I) An OpenMP Parallel Sections and Loop Example

```
void parfoo( )
{   int m, y, x[5000];
    float w, z[3000];
#pragma omp parallel sections shared(w, z, y, x)
   {
        w = floatpoint_foo(z, 3000);
    #pragma omp section
        y = myinteger_goo(x, 5000) ;
   }
#pragma omp parallel for private(m) shared(y, z, w) schedule(guided)
   for (m=0; m<3000; m++) {
        z[m] = z[m] * w * y;
   }
}
```

Essentially, the multithreaded code generator inserts the thread invocation call *__kmpc_fork_call*(…) with *T-entry* point and data environment (source line information *loc*, thread number *tid*, etc.) for each parallel loop, parallel sections or parallel region, and transforms a serial loop, sections, or region to a multithreaded loop, sections, or region, respectively. In this example, the *pre-pass* first converts a *parallel section* to a *parallel loop*. Then, the multithreaded code generator localizes loop lower-bound and upper-bound, privatizes the section id variable, and generates runtime initialization and synchronization code such as the call *__kmpc_static_init*(…) and the call *__kmpc_static_fini*(…) for the T-region marked with [T_entry, T-ret] nodes. For the parallel loop in the routine "*parfoo*" with the scheduling type *guided*, the OpenMP parallelization involves (i) generating a runtime dispatch and initialization routine (*__kmpc_dispatch_init*) call to pass necessary information to the runtime system; (ii) generating an enclosing while loop to dispatch *loop-chunk* at runtime through the *__kmpc_dispatch_next* routine in the library; (iii) localizing the loop lower-bound, upper-bound, and privatizing the loop control variable '*m*.'

(E1-II) Pesudo Multithreaded Code after Parallelization

```
R-entry void parfoo( )
{   int m, y, x[5000];
    float w, z[3000];
   __kmpc_fork_call(loc, 4, T-entry(_parfoo_psection_0), &w,z,x,&y)
   goto L1:
   T-entry _parfoo_psection_0(loc, tid, *w, z[], *y, x[]) {
      lower_pid = 0;
      upper_pid = 1;
     __kmpc_static_init(loc, tid, STATIC, &lower_pid, &upper_pid...);
     for (pid=lower_pid, pid<=upper_pid; pid++) {
         if (pid == 0) {
            *w = floatpoint_foo(z, 3000);
         } else if (pid == 1) {
            *y = myinteger_goo(x, 5000);
         }
     }
    __kmpc_static_fini(loc, tid);
    T-ret;
  }
L1:
   __kmpc_fork_call(loc, 3, T-entry(_parfoo_ploop_1), &w, z, &y);
  goto L2:
  T-entry _parfoo_ploop_1(loc, tid, *w, z[], *y) {
     lower = 0;
     upper = 3000;
       __kmpc_dispatch_init(loc, tid, GUIDED, &lower, &upper, …);
      while (__kmpc_dispatch_next(loc, tid, &lower, &upper, …)) {
         for (prv_m=lower; prv<upper; prv_m++) {
            z[prv_m] = z[prv_m] * (*w) * (*y);
         }
```

```
    }
    T-ret;
  }
L2:
  R-return;
}
```

There are four well-defined properties of the T-region graph model of the *Multi-Entry Threading* technique:

1. T-region is a sub-graph on top of the pragma-aware control-flow graph, which is identified by the T-entry and T-ret node for a parallel region, sections, or loop.
2. T-region can be nested to present the hierarchy of nested parallelism, e.g., T-region(*k*) = [T-entry(*m*), T-region(*m*), T-ret(*m*)], where the '*k*' and '*m*' are the unique id of a T-region inside this routine.
3. T-region shares the same local memory locations of all local static variables of R-entry (Routine entry). In other words, the local static variables are visible by every T-region associated with this routine.
4. Multiple T-regions are permitted to represent multiple parallel constructs at the same nesting level.

With the T-region graph representation of the Multi-Entry Threading technique, the OpenMP parallelizer and low-level code generators do not generate a separate routine (or compilation unit) for a parallel region or parallel loop. All newly generated multithreaded code blocks (T-regions) for parallel loops are still kept inlined within the same compilation unit. The code transformations are done in a natural way.

From a compiler engineering point of view, the Multi-Entry Threading technique greatly reduces the complexity of generating separate routines in the Intel compiler. In addition, this technique minimizes the impact of OpenMP parallelization on well supported optimizations in the Intel compiler such as constant propagation, vectorization, PRE, PDSE, scalar replacement, loop transformation, interprocedural optimization, and profile-feedback guided optimization (PGO). This meets one of the design goals, namely, to tightly incorporate parallelization with all well-known and advanced compiler optimizations in the Intel compiler.

DATA-SHARING AND PRIVATIZATION

When a routine calls another routine, the communication between them is through global variables and through arguments of the called routine (or callee). This argument-passing across the routine boundary introduces some overhead. The more arguments are passed, the more overhead is introduced. If the call-by-reference method is used for associating actual and dummy arguments, the caller passes to the callee the storage address of the actual argument, and the reference to the dummy argument in the callee becomes an indirect reference. Many optimizations could become disabled by this memory de-referencing. Given the Guide run-time library API, with the *outlining* technology, the parallelizer needs to create a separate routine for a parallel construct, which means the address of each local static variable has to be passed to the outlined routine, since the local static variables in a routine are not visible to other routines. Thus, there are three drawbacks with the *outlining* technique [5, 6]: (i) it adds extra overhead due to argument-passing to outlined routine for sharing local static variables among threads; (ii) it causes less efficient memory access due to memory de-referencing in the outlined routine; and (iii) it may disable some optimizations such as Intra-Register Vectorization and Partial Redundancy Elimination (PRE).

In our implementation of OpenMP parallelization, we are able to overcome these drawbacks based on our Multi-Entry Threading technique. The advances of our technique are: (i) the extra overhead of sharing local static variables is reduced to zero; (ii) no extra memory de-referencing is introduced for accessing local static shared variables; and (iii) later scalar optimizations on local static variables are preserved. The following (E2-I) example has local static variables '*w*,' '*z*,' '*y*,' and '*x*' that are marked as *shared*.

(E2-I) An OpenMP Parallel Sections Example

```
void staticparfoo( )
{  int m;
   static int    y, x[5000];
   static float  w, z[5000];
#pragma omp parallel sections shared(w, z, y, x)
  {
      w = floatpoint_foo(z, 5000);
   #pragma omp section
      y = myinteger_goo(x, 5000) ;
  }
  return;
}
```

In (E2-II), we show the C-like pseudo-multithreaded code generated by the parallelizer. As we see, there are no extra arguments on the *T-entry* node for sharing local static variable '*w*,' '*z*,' '*y*,' and '*x*,' and there is no pointer de-referencing inside the *T-region* for sharing those local static variables among all threads in the team.

(E2-II) Pesudo Multithreaded Code after Parallelization

```
R-entry void staticfoo( )
{   int m;
    static int    y, x[5000];
    static float  w, z[5000];
   __kmpc_fork_call(loc, 0, T-entry(_staticfoo_psection_0))
   goto L1:
   T-entry _parfoo_psection_0(loc, tid) {
      lower_pid = 0; upper_pid = 1;
```

```
      __kmpc_static_init(loc,  tid,  STATIC,  &lower_pid,
&upper_pid...);
      for (pid=lower_pid,  pid<=upper_pid; pid++)  {
          if (pid == 0) {
              w = floatpoint_foo(z, 5000);
          } else if (pid == 1) {
              y = myinteger_goo(x, 5000);
          }
      }
      __kmpc_static_fini(loc, tid);
      T-ret;
   }
L1:    R-return;
}
```

It is well known that the privatization technique can break cycles in a dependence graph and eliminate loop-carried dependencies, so parallelization can be enabled effectively. Actually, privatization removes memory de-references as well. There are three privatization clauses: *firstprivate*, *lastprivate* and *private*, defined in the OpenMP Fortran and C++ standard. Given an OpenMP Fortran example in (E3-I), we see that variables '*x*' and '*y*' are marked as *firstprivate*. The intermediate code before parallelization contains memory de-references *(F32 *)*x* and *(F32 *)*y* (where 'F32' indicates 32-bit floating-point data type) for accessing the variables '*x*' and '*y*' in terms of the call-by-reference argument-passing method used in the Fortran language, as shown in (E3-II).

(E3-I) An OpenMP Fortran Example

```
      subroutine privatefoo(x, y)
      real x, y
      real, save :: a(100)
!$omp parallel do firstprivate(x,y) shared(a)
      do k=1, 100
        a(k) = x + y*k
      end do
      return
      end
```

(E3-II) Pesudo Intermediate Code before Parallelization

```
R-entry void  privatefoo(x, y)
{   … … …
    DIR_OMP_PARALLEL_LOOP      FIRSTPRIVATE(x,      y)
SHARED(a)
    k = 1;
 L3:
    a[k] = *(F32*)x + *(F32*)y * k;
    k = k + 1;
    if (k <= 100 )  { goto L3;  }
    DIR_OMP_END_PARALLEL_LOOP
    R-return
}
```

(E3-III) Pesudo Multithreaded Code after Parallelization

```
R-entry void  privatefoo(x, y)
{    … … …
    __kmpc_fork_call(loc, 2, T-entry(_privatefoo_ploop_0), x, y)
    goto L1:
    T-entry _privatefoo_ploop_0(loc, tid, *x, *y) {
        lower = 0;
        upper = 99;
        prv_x = *(F32 *)x;
        prv_y = *(F32 *)y;
       __kmpc_static_init(loc, tid, STATIC, &lower, &upper, ...);
        prv_k = lower;
        L4:
            a[prv_k] = prv_x + prv_y * prv_k;
            prv_k = prv_k + 1
            if (prv_k <= upper) { goto L4: }
      __kmpc_static_fini(loc, tid);
      T-ret;
   }
L1:    R-return;
}
```

As we can see from (E3-III), privatization has eliminated the memory de-references **x* and **y* inside the parallel loop through the pre-load and pre-copy into the local stack variables '*prv_x*' and '*prv_y*' created by the privatizer. Obviously, this transformation improves the performance by lifting memory de-references outside the loop.

ADVANCED OPTIMIZATIONS

In order to fully leverage advanced scalar optimizations before and after the OpenMP parallelization phase, an optimization phase ordering is carefully designed and implemented in the Intel compiler. In this section, we discuss our design relative to advanced optimizations such as Inter-Procedural Optimization (IPO) [9] and Partial Redundancy Elimination (PRE).

The IPO phase is enabled before OpenMP parallelization at the higher optimization level, so that the Profile-feedback Guided Optimization (PGO), inlining, partial inlining, and forward-substitution can use and benefit from all heuristic and profiling information without any disturbance from multithreaded code generated by the OpenMP parallelizer. In this way, the parallelization is done based on the optimized code. See example E4-I.

(E4-I) An OpenMP Example for Using IPO

```
float w;
void floatpoint_add(float z[ ], int n)
{  int k;
#pragma omp for reduction(+: w) private(k)
   for (k =0; k < n; k++) {
       w = w + z[k];
   }
}

void  inlinefoo( )
{   static float  w, z[5000];
#pragma omp parallel shared(w, z)
   {
       floatpoint_add(z, 5000);
   }
}
```

(E4-II) Pesudo intermediate code after IPO

```
float  w;
R-entry void  inlinefoo( )
{  static float  w, z[5000];
#pragma omp parallel shared(w, z)
   {    int k;
#pragma omp for reduction(+: w) private(k)
        for (k =0; k < 5000; k++) {
           w = w + z[k];
        }
   }
   R-return;
}
```

With IPO inlining and forward-substitution optimization, the subroutine '*floatpoint_add*' is inlined to the subroutine '*inlinefoo*,' the variable '*n*' is substituted with the constant 5000. If the IPO is enabled after OpenMP parallelization, then inlining and forward-substitution may not be able to kick in due to extensive code transformation within the '*floatpoint_add*' and '*inlinefoo*' by the parallelization phase, and due to the changes of the profiling information.

The PRE phase was implemented based on the algorithm in [7] and runs after OpenMP parallelization. Given the example E5-I, the expression '*x+y*k*' is redundant and only needs to be evaluated once for each iteration, and '*x*y*' can be lifted outside the parallel loop.

(E5-I) An OpenMP for Using PRE

```
int b[200], c[200];
void prefoo(int x, int y)  /* x=1 and y=2  in caller */
{ int a[100], k;

#pragma omp parallel for private(k) shared(a, b, c, x,  y)
   for (k = 0; k < 100; k++) {
       a[k] = b[x + y*k] + c[x+y k] + x*y;
  }
  return;
}
```

(E5-II) Pesudo Multithreaded Code from Parallelization and PRE

```
R-entry void prefoo(int x, int y)
{    … … …
    __kmpc_fork_call(loc, 2, T-entry(_prefoo_ploop_0), &x, &y)
    goto L1:
    T-entry _privatefoo_ploop_0(loc, tid, *x, *y) {
       lower = 0;
       upper = 99;
       prv_x = *(SI32 *)x;
       prv_y = *(SI32 *)y;
       t0 = prv_x * prv_y;
      __kmpc_static_init(loc, tid, STATIC, &lower, &upper, ...);
       L3:
            t1 = prv_x + prv_y * prv_k;
            a[prv_k] = b[t1] + c[t1] + t0;
            prv_k = prv_k + 1
            if (prv_k <= upper) {
                goto L3:
            }
      __kmpc_static_fini(loc, tid);
      T-ret;
   }
L1:
   R-return;
}
```

Redundancy is removed through saving the value of the redundant expression in a temporary variable and later reusing that value instead of reevaluating the expression. However, we must be careful with moving code around parallel constructs, since it could generate an unsafe insertion of code for a lifted common expression without knowing the parallel region or parallel loop boundary. Our solution is to apply PRE within each *T-region* after OpenMP parallelization. This guarantees that the correct code is generated. In the code example shown above, we see that '*t0*' and '*t1*' are created as register temporary variables. The 't0' is lifted outside the parallel loop, but it is inserted within the *T-region* and only evaluated once for each thread. The '*t1*' is only evaluated once for each loop iteration. In our experience, there is almost no difference between this and applying PRE optimization to sequential code. There are many more design and implementation details related to incorporating advanced optimizations with parallelization. In the next section, we discuss how the OpenMP parallelization incorporates intra-register vectorization to effectively exploit multi-level parallelism.

MULTI-LEVEL PARALLELISM

The SIMD extensions to the Intel Architecture provide an alternative way to utilize data parallelism in multi-media and scientific applications. These extensions let multiple functional units operate simultaneously on packed data elements, i.e., relatively short vectors that reside in memory or registers. The Pentium 4 processor features the streaming-SIMD-extensions (SSE and SSE2) that support floating-point operations on 4 packed single-precision and 2 packed double-precision floating-point numbers, as well as integer operations on 16 packed bytes, 8 packed words and 4 packed dwords. The Intel compiler supports the automatic conversion of serial loops into SIMD form, a transformation that we refer to as intra-register vectorization [3,4].

Combining intra-register vectorization with parallelization for hyper- or multithreading enables the exploitation of multi-level parallelism, i.e., using the different forms of parallelism that are present in a code fragment to obtain high performance. Take, for instance, the code for matrix-vector multiplication shown in example (E6-I).

(E6-I) An OpenMP-Vector Loop Example

```
double a[N][N], x[N], y[N];
…
#pragma omp parallel for private(k,j)
 for (k = 0; k < N; k++) { /* parallel loop */
   double d = 0.0;
   for (j = 0; j < N; j++) { /* vector loop */
       d += a[k][j] * y[j];
   }
   x[k] = d;
 }
…
```

(E6-II) Pseudo code after Parallelization and Vectorization

```
    __kmpc_fork_call(loc, 0, T-entry(_ompvec_ploop_0),  … )
    goto L1:
    T-entry _ompvec_ploop_0(loc, tid) {
       lower = 0;
       upper = N;
      __kmpc_static_init(loc, tid, STATIC, &lower, &upper, ...);
       prv_k = lower;
L2:
       xorpd    xmm0, xmm0            ; reset accumulator
L3:
       movapd   xmm1, _a[ecx+edx]  ; load 2 DP  from a
       mulpd    xmm1, _y[edx]         ; mult 2 DP  from y
       addpd    xmm0, xmm1            ; add  2 DP  into accumulator
       add      edx, 16               ;
       cmp      edx, eax               ;
       jl        L3                    ; looping logic

       movapd    xmm1, xmm0          ;
       unpckhpd xmm1, xmm1           ;
       addsd     xmm0, xmm1          ; compute final sum

        store result in  x[prv_k]

        prv_k = prv_k + 1
        if (prv_k <= upper) goto L2;

       __kmpc_static_fini(loc, tid);
       T-ret;
    }
L1:  … …
```

In this example, parallelism appears at multiple levels. The iterations of the outermost *k-loop* may execute independently, as has been made explicit with an OpenMP pragma. The reduction performed in the innermost *j-loop* provides yet another level of parallelism. This loop can be implemented by accumulating partial sums in SIMD style, followed by code that constructs the final sum. In (E6-II), we illustrate how these two levels of parallelism can be exploited (where we assume that all access patterns in the vector loop are aligned at a 16-byte boundary).

If the alignment of memory references cannot be determined at compile-time, the Intel compiler has at its disposal several alignment optimizations (such as run-time loop peeling) to avoid performance penalties that are usually associated with unaligned memory accesses. Dynamic data dependence testing is used to allow the compiler to proceed with vectorization in situations where analysis has failed to prove independence statically. These advanced techniques (and others) have been discussed in detail in previous work [4].

PERFORMANCE EVALUATION

The performance study of SPEC OpenMP benchmarks is carried out on a pre-production 1-CPU Hyper-Threading Technology-enabled Intel Xeon processor system running at 1.7GHz, with 512M memory, an 8K L1-Cache, and a 256K L2-Cache. All benchmarks and applications studied in this paper are compiled by the Intel OpenMP C++/Fortran compiler. For the performance study, we chose a subset of SPEC OMPM2001 benchmarks to demonstrate the performance effect of Hyper-Threading Technology. The SPEC OMPM2001 is a benchmark suite that consists of a set of scientific applications. Those SPEC OpenMP benchmarks target small and medium scale (2- to 16-way) SMP multiprocessor systems and the memory footprint reaches 1.6GB for several very large application programs.

The performance scaling is derived from serial execution (SEQ) with Hyper-Threading Technology disabled, and multithreaded execution under one thread and two threads with Hyper-Threading Technology disabled and enabled. In Figure 3, we show the normalized speed-up of the chosen OpenMP benchmarks compared to the serial execution with Hyper-Threading Technology disabled. The OMP1 and OMP2 denote the multithreaded code generated by the Intel OpenMP C++/Fortran compiler executing with one thread and two threads, respectively.

As we see, the multithreaded code generated by the Intel compiler on a Hyper-Threading Technology-enabled Intel Xeon processor 1-CPU system achieved a performance improvement of 4% to 34% (OMP2 w/ HT). The 320.*equake* obtained a 14% performance gain from scalar optimizations enabled by OpenMP (OMP1 w/o HT).

Another 20% performance improvement was achieved by the second thread running on the second logical processor, resulting in a 34% performance gain overall (OMP2 w/ HT). The multithreaded code of the 330.*art* does not show OpenMP overhead, and obtained an 8% speed-up. A 23% slowdown was observed from the 332.*ammp* due to the overhead of thread creation, forking, synchronization, scheduling at run-time, and memory access de-referencing for sharing local stack variables (OMP1 w/o HT), but the second thread running on the second logical processor contributed to the overall 4% performance improvement (OMP2 w/ HT).

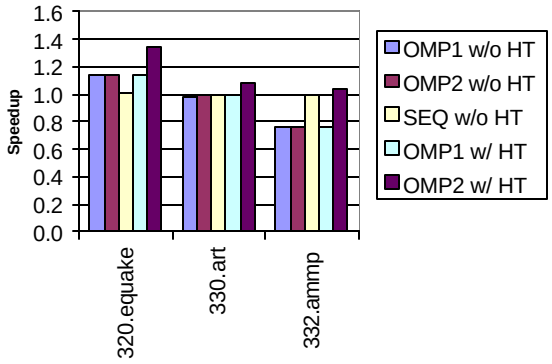

Figure 3: Performance of OpenMP benchmarks

In Figure 4, we show the performance speed-up of four image-processing functions taken from the OpenMP version IPPI library developed by the Intel Performance Library group. The performance speed-up ranges from 1.26x to 1.41x (image size 720x480) on a pre-production Hyper-Threading Technology-enabled Intel Xeon processor 1-CPU system running at 1.8GHz, with 512M of memory, an 8K L1-Cache and a 256K L2-Cache.

As far as we know, there are around 300 image-processing and JPEG functions multithreaded by OpenMP directives in the Intel IPPI performance library. An average speedup of 1.4x was reported when compared with the serial execution of those routines on a pre-production Intel 1.8GHz Hyper-Threading Technology -enabled Intel Xeon Processor 1-CPU system.

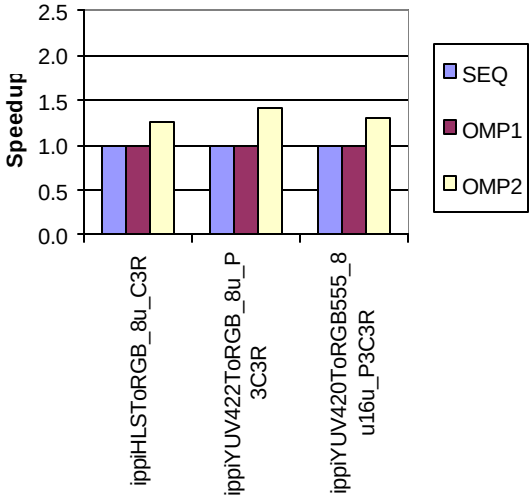

Figure 4: Performance of image processing functions

In Figure 5, we show some performance results for the matrix-vector multiplication kernel discussed earlier on a pre-production Hyper-Threading Technology-enabled Intel Xeon processor dual-CPU system running at 1.5GHz with 512MB of memory, an 8K L1-Cache and a 256K L2-Cache. This graph shows speed-ups (relative to serial execution) for varying matrix sizes for vector execution (VEC), multithreaded execution using two threads and four threads, (OMP2) and (OMP4), respectively, and vector-multithreaded execution using two and four threads, (OMP2+VEC) and (OMP4+VEC), respectively.

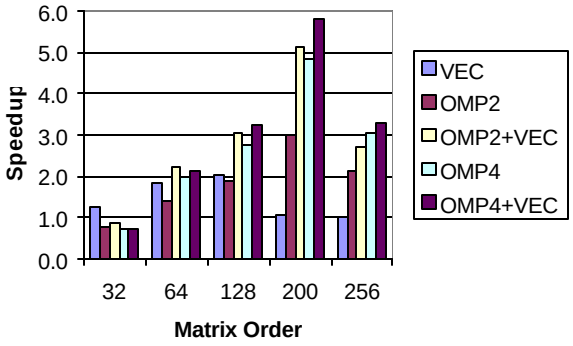

Figure 5: Performance of Matrix x Vector kernel

Timings were obtained by calling the kernel many times and dividing the total execution time accordingly, which implies that for the data sets that completely fit in cache, the kernel is computationally bound. In these cases, intra-register vectorization alone obtains a speed-up of up to 2x. For the larger data sets, where the kernel becomes more memory bound, the improvements of merely intra-register vectorization become less evident. As we have seen

before, the overhead associated with multithreading causes a slight slowdown for the matrix size 32x32. For the larger matrices ranging from 64x64 to 256x256, the relative overhead introduced by parallelization becomes negligible and observed speed-up ranges from 1.4x to 5.8x.

The difference between (OMP2) and (OPM4) for matrix size 200x200 reveals a 1.6x performance gain. For the same matrix size, the performance gain from the versions that are optimized with intra-register vectorization, (OMP2+VEC) and (OMP4+VEC), is 1.2x. The best performance gains are obtained when all levels of parallelism (SIMD parallelism and parallelism due to Hyper-Threading Technology and multithreading) are exploited simultaneously, yielding a speed-up of up to 5.8x with four threads (OMP4+VEC) and a speed-up of 5.1x with two threads (OMP2+VEC).

CONCLUSION

With the growing processor-memory performance gap, memory latency becomes a major bottleneck for achieving high performance for various applications. There are a number of multithreading techniques proposed to hide memory latency. Intel's Hyper-Threading Technology is a very promising technology that allows a single processor to manage data as if it were two processors by executing data instructions in parallel rather than serially. With this new technology, the performance of applications can be greatly improved by exploiting thread-level parallelism. The potential gains are only obtained, however, if an application program is multithreaded. The Intel OpenMP C++/Fortran compiler has been designed to leverage the rich set of performance enabling features, such as Hyper-Threading Technology and the Streaming-SIMD-Extensions (SSE and SSE2), this is achieved by tightly integrating OpenMP directive- or pragma-guided parallelization with other well-known and advanced optimizations to generate efficient multithreaded code for exploiting parallelism at various levels. The results of performance measurement show that OpenMP applications compiled with the Intel C++/Fortran compiler can achieve great performance gains on Intel single and multiprocessor systems that are enabled with Hyper-Threading Technology.

ACKNOWLEDGMENTS

The authors thank the other members of the compiler team for their great work in implementing the Intel high-performance C++/Fortran compiler. In particular, we thank Max Domeika for the OpenMP C++/C front-end support, Michael L. Ross and Bhanu Shankar for the OpenMP Fortran front-end support, Knud J. Kirkegaard for IPO support, and Zia Ansari for PCG support. Special thanks go to Sanjiv Shah and the compiler group at KSL for providing the Guide runtime library, and to the INNL library team for providing the Short Vector Mathematical Library. Both libraries are currently part of the Intel C++/Fortran compiler. Many thanks go to Boris Sabanin for providing the performance numbers for Intel IPPI Image processing functions.

REFERENCES

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, *Compilers–Principles, Techniques and Tools*, Addison-Wesley Publishing Company, Boston, Massachusetts, 1986.

[2] Vishal Aslot, et .al., "SPEComp: A New Benchmark Suite for Measuring Parallel Computer Performance," in Proceedings of WOMPAT 2001, *Workshop on OpenMP Applications and Tools, Lecture Notes in Computer Science*, 2104, pages 1-10, July 2001.

[3] Aart Bik, Milind Girkar, Paul Grey, and Xinmin Tian, "Efficient Exploitation of Parallelism on Pentium® III and Pentium® 4 Processor-Based Systems," *Intel Technology Journal*, Q1 2001, http://intel.com/technology/itj/q12001/articles/art_6.htm.

[4] Aart Bik, Milind Girkar, Paul Grey, and Xinmin Tian, "Automatic Intra-Register Vectorization for the Intel® Architecture," accepted by the *International Journal of Parallel Programming*, December, 2001.

[5] C. Brunschen and M. Brorsson, "OdinMP/CCp–A Portable Implementation of OpenMP for C," in *Proceedings of the First European Workshop on OpenMP (EWOMP)*, September. 1999.

[6] Jyh-Herng Chow, Leonard E. Lyon, and Vivek Sarkar, "Automatic Parallelization for Symmetric Shared-Memory Multiprocessors, in *Proceedings of CASCON'96: 76-89*, Toronto, ON, November 12-14, 1996.

[7] F. Chow, S. Chan, R. Kennedy, S. Liu, R. Lo, and P. Tu, "A new algorithm for partial redundancy elimination based on SSA form," in *Proceedings of the ACM*

SIGPLAN '97 Conference on Programming Language Design and Implementation, June 1997, pp. 273-286.

[8] Carole Dulong, "The IA-64 Architecture at Work," *IEEE Computer*, July 1998, pp. 24-32.

[9] Mary W. Hall, Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, and Monica S. Lam, "Detecting Coarse-Grain Parallelism Using an Interprocedural Parallelizing Compiler," in *Proceedings of Supercomputing*, San Diego, California, Dec. 1995.

[10] Michael J. Wolfe, *High Performance Compilers for Parallel Computers*, Addison-Wesley Publishing Company, Redwood City, California, 1996.

[11] Intel Corporation, *Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture*, Intel Corporation, 2001, http://developer.intel.com/

[12] OpenMP Architecture Review Board, "OpenMP C and C++ Application Program Interface," Version 1.0, October 1998, http://www.openmp.org

[13] OpenMP Architecture Review Board, "OpenMP Fortran Application Program Interface," Version 2.0, November 2000, http://www.openmp.org

[14] Debbie Marr, et al., "Hyper-Threading Technology Architecture and Microarchitecture," *Intel Technology Journal*, Q1 2002.

AUTHORS' BIOGRAPHIES

Xinmin Tian is currently working in the vectorization and parallelization group at Intel Corp. where he works on compiler parallelization and optimization. He manages the OpenMP Parallelization group. He holds B.Sc., M.Sc., and Ph.D. degrees in Computer Science from Tsinghua University. He was a postdoctoral researcher in the School of Computer Science at McGill University, Montreal. Before joining Intel Corp., he worked on a parallelizing compiler, code generation, and performance optimization at IBM. His e-mail is xinmin.tian@intel.com

Aart Bik received his M.Sc. degree in Computer Science from Utrecht University, The Netherlands, in 1992 and his Ph.D. degree from Leiden University, The Netherlands, in 1996. In 1997, he was a postdoctoral researcher at Indiana University, Bloomington, Indiana, where he conducted research in high-performance compilers for Java*. In 1998, he joined Intel Corporation where he is currently working in the vectorization and parallelization group. His e-mail is aart.bik@intel.com

Milind Girkar received a B.Tech. degree from the Indian Institute of Technology, Mumbai, an M.Sc. degree from Vanderbilt University, and a Ph.D. degree in Computer Science from the University of Illinois at Urbana-Champaign in Computer Science. Currently, he manages the IA-32 Compiler Development group. Before joining Intel Corp., he worked on an optimizing compiler for the UltraSPARC platform at Sun Microsystems. His e-mail is milind.girkar@intel.com

Paul Grey did his B.Sc. degree in Applied Physics at the University of the West Indies and his M.Sc. degree in Computer Engineering at the University of Southern California. Currently he is working at Intel Corp. on compiler optimizations for parallel computing. Before joining Intel, he worked on parallel compilers, parallel programming tools, and graphics system software at Kuck and Associates, Inc., SUN, and SGI. He is interested in optimizing compilers, advanced microarchitecture, and parallel computers. His e-mail is paul.grey@intel.com

Hideki Saito received a B.E. degree in Information Science in 1993 from Kyoto University, Japan, and a M.S. degree in Computer Science in 1998 from University of Illinois at Urbana-Champaign, where he is currently a Ph.D. candidate. He joined Intel Corporation in June 2000 and has been working on multithreading and performance analysis. He is a member of the OpenMP Parallelization group. His e-mail is hideki.saito@intel.com

Ernesto Su received a B.S. degree from Columbia University, and M.S. and Ph.D degrees from the University of Illinois at Urbana-Champaign, all in Electrical Engineering. He joined Intel Corp. in 1997 and is currently working in the OpenMP Parallelization group. His research interests include compiler performance optimizations, parallelizing compilers, and computer architectures. His e-mail is ernesto.su@intel.com.

*Other brands and names are the property of their respective owners.