Compiler and Runtime Support for Running OpenMP Programs on Pentium- and Itanium-Architectures

Xinmin Tian[1], Milind Girkar[1], Sanjiv Shah[2], Douglas Armstrong[2], Ernesto Su[1], Paul Petersen[2]
[1]Intel Compiler Laboratory, Software Solution Group, Intel Corporation
[1]3600 Juliette Lane, Santa Clara, CA 95052, USA
[2]KAI Software Laboratory, Software Solution Group, Intel Corporation
[2]1906 Fox Drive, Champaign, IL 61820, USA
{Xinmin.Tian, Milind.Girkar, Sanjiv.Shah, Douglas.Armstrong, Ernesto.Su, Paul.Petersen}@intel.com

Abstract

Exploiting Thread-Level Parallelism (TLP) is a promising way to improve the performance of applications with the advent of general-purpose cost effective uni-processor and shared-memory multiprocessor systems. In this paper, we describe the OpenMP implementation in the Intel® C++ and Fortran compilers for Intel platforms. We present our major design consideration and decisions in the Intel compiler for generating efficient multithreaded codes guided by OpenMP directives and pragmas. We describe several transformation phases in the compiler for the OpenMP* parallelization. In addition to compiler support, the OpenMP runtime library is a critical part of the Intel compiler. We present runtime techniques developed in the Intel OpenMP runtime library for exploiting thread-level parallelism as well as integrating the OpenMP support with other forms of threading termed as sibling parallelism. The performance results of a set of benchmarks show good speedups over the well-optimized serial code performance on Intel® Pentium- and Itanium-processor based systems.*

Keywords: Parallelization, Hyper-Threading technology, OpenMP, compiler optimization, thread-level parallelism, shared-memory multiprocessor

1. Introduction

The explicitly parallel computing technology behind the OpenMP shared-memory programming model [2,3,5,6] provides a rich set of features, which allow the compiler to exploit thread-level parallelism and optimize applications on Intel Architecture (IA) based platforms. Shifting most of the complex tasks from the programmer to the compiler encourages programmers to write and port program to fully take advantage of the available state-of-the-art architecture features, such as Hyper-Threading technology, to exploit thread-level parallelism and boost application performance

The Intel C++ and Fortran compilers support the OpenMP pragmas and directives in languages C++/C and Fortran95, on Windows and Linux platforms and on IA-32 [11] and Itanium Processor Family (IPF) architectures.

The Intel OpenMP implementation in the compiler strives to: (a) generate multithreaded code which gains a true speedup over optimized uniprocessor code, (b) integrate parallelization tightly with advanced interprocedure, scalar and loop optimizations such as intra-register vectorization [2, 4] and memory hierarchy oriented optimizations [9, 10] to achieve better cache locality and efficiently exploit multi-level parallelism, and (c) minimize the overhead of data-sharing among threads. In this paper, we describe the implementation of the parallelization phase in the Intel compiler for OpenMP support. The remainder of this paper is organized as follows. The Section 2 presents an overview of the Intel high-performance compiler. Section 3 describes design decisions made in the Intel C++/Fortran compiler for generating efficient multithreaded code guided by OpenMP directives or pragmas, including code transformation phases where the OpenMP parallelizer interacts tightly with other optimizations. Section 4 gives a high-level overview of the software architecture of the OpenMP run-time library and presents the key features and techniques developed in the Intel OpenMP run-time library for the Intel compiler. The Section 5 show performance results of a set of benchmarks on IA-32 and IA-64 based platforms. Finally, concluding remarks can be found in Section 6.

2. Compiler Overview

The Intel C++ and Fortran compilers have a single common intermediate representation named IL0 for the C++/C and Fortran95 languages. Hence, the OpenMP directive-guided parallelization, as well as a majority of other optimizations, is applicable through a single high-level transformation [2] irrespective of the high-level source language. Throughout the rest of this paper, we refer to the Intel C++ and Fortran compilers for Intel Pentium and Itanium processor family architectures collectively as "the Intel compiler". In order to establish the context in which the OpenMP parallelization works, we give a brief overview of the Intel compiler for Pentium and Itanium processor-based platforms.

Inter-Procedural Optimization (IPO): this component includes points-to analysis and mod/ref analysis required by many other optimizations. Points-to analysis expands the capabilities of memory disambiguation by determining that which memory locations may be referenced by a memory reference.

Multi-Entry Threading (MET): we have developed and implemented the new compiler technology named *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 or loop [1,2].

Multi-Level Parallelism (MLP): Intel compiler supports intra-register vectorization for Pentium family processor [2], and software pipelining for Itanium family processor for exploiting instruction-level parallelism (ILP) on top of exploiting thread-level parallelism (TLP). Exploiting MLP (TLP+ILP) ensures the compiler fully utilizes the rich set of performance features of Intel architecture for achieving the highest application performance.

High-Level Optimization (HLO): those optimizations in HLO include loop transformations such as loop fusion, loop tiling, loop unroll-and-jam, loop distribution, profile-guided data prefetching, scalar replacement and data optimizations to improve data locality and reduce memory access latency.

Other Scalar Optimization Components: Intel compiler implements an extensive set of scalar optimizations such as branch-merging, strength reduction, constant propagation, dead code elimination, copy propagation, partial dead store elimination, and partial redundancy elimination (PRE) [8].

Architecture-specific code generation components include instruction scheduling, register allocation, code ordering, advanced instruction selection, and global code scheduling.

3. Implementation

In order to support the OpenMP programming model, the Intel compiler has been extended throughout its various components. First, the IL0 intermediate representation was extended to represent the OpenMP directives/pragmas and clauses. The compiler front-end parses OpenMP directives (or pragmas) to generate consistent IL0 representation of OpenMP code for the compiler middle-end. The OpenMP parallelizer generates multithreaded codes based on IL0 codes corresponding to OpenMP constructs.

The design philosophy behind the implementation of the OpenMP programming model in the Intel compiler is that a single OpenMP parallelizer implementation is used across all languages (C++/C and Fortran 95) and architectures (IA-32 and IPF). The Intel compiler generates multithreaded code that has references to a high-level run-time library API designed and developed at Intel KAI Software Laboratory. The following sections describe several transformations and optimizations for OpenMP parallelization.

3.1 Compiler Front-End Support

The compiler's front-end generates an IL0 representation of the OpenMP code as shown in Figure 1, where the *for* loop has been lowered into *if* and *goto* statements after the IL0 lowering phase. Each OpenMP pragma has been converted into an equivalent pair of IL0 directive and its matching end directive, which helps the WRN (work-region-node) graph builder of the OpenMP parallelizer define the boundaries of the OpenMP constructs.

Besides syntax and semantics checking, one of the issues the FE needs to address is finding the implicit attributes of variables that are not explicitly listed in a clause. In this example, the array '*a*' and induction variable '*k*' are listed as *shared* and *private*, respectively. However, the array '*b*' and variable '*x*' are not specified in any clause. Based on the OpenMP specification, the FE treats a locally declared automatic variable as a private variable of the OpenMP construct that immediately encloses it lexically. Thus, the variable *x* is added to the private list of the worksharing *for* construct.

```
void parwork() /* OpenMP C code sample */
{  double a[1000], b[1000];
   int       k;
#pragma omp parallel shared(a) private(k)
   {    int x;
#pragma omp for schedule(dynamic)
        for  (k=0; k<16; k++)  {  do_work(k, a, b, &x);  }
   }
}
entry extern void  _parwork()  /* IL0 pseudo-code after Front-End */
{  DIR_PARALLEL QUAL_SHARED_VAR (a) QUAL_PRIVATE_VAR (k)
                QUAL_SHARED_VAR (b) QUAL_PRIVATE_VAR (x)
      DIR_LOOP  QUAL_SCHEDULE  (DYNAMIC)
       k = 0(SI32);                  /* SI32 denotes the 32-bit signed int type */
  L13:  if ( k < 16(SI32) )    {
              t0 = _do_work( k, &a[0(SI32)], &b[0(SI32)], &x);
              k = k + 1(SI32);
              goto L13;
           }
      DIR_END_LOOP
  DIR_END_PARALLEL
  return ;
}
```

Figure 1. Parallel Region and Worksharing Loop Example

Next step, the FE finds the implicit shared variables of the parallel region based on a rule in OpenMP specification -- "the default attribute is *default shared*" if the *default* clause is not specified. The results of the analysis is that the FE generates private(*x*) and shared(*b*) for the parallel construct. Note that register temps (e.g. *t0* in Figure 1) created by the FE are treated as private in the BE.

3.2 Pre-Pass Transformation

The pre-pass performs the transformation that converts a *parallel section* to a parallel *for* loop, so the implementation of parallel *sections* construct can leverage the multithreaded

code generation of the parallel loop. Essentially, a parallel loop is generated and the loop trip count is the number of sections. In Figure 2(a), there are three sections inside a *parallel sections* construct, the pre-pass creates a parallel loop with trip count 3, see Figure 2(b).

```
void parsectfoo()
{  int     y,  x[5000];
   float   w, z[3000];
   double u, v[5000];

#pragma omp parallel sections shared(w, z, y, x, u, v)
     {
          w = floatpoint_sect(z, 3000);
     #pragma omp section
          y = myinteger_sect(x, 5000);
     #pragma omp section
          u = mydouble_sect(v, 5000);
     }
  }
```

(a) parallel sections before pre-pass

```
void parsectfoo()
{  int     y,  x[5000];
   float   w, z[3000];
   double u, v[5000];
 DIR_PAR_LOOP QUAL_SHARED(w, z, y, x, u, v) QUAL_PRIVATE(i
     for (id=1; id<=3; id++) {
         switch (id) {
             case 1: w = floatpoint_sect(z, 3000);  break;
             case 2: y = myinteger_sect(x, 5000);  break;
             case 3: u = mydouble_sect(v, 5000);  break;
         }
     }
 DIR_END_PAR_LOOP
}
```

(b) generated parallel loop after pre-pass

Figure 2. Pseudo-code After Pre-Pass of Parallelization

Given that the granularity of the parallel *sections* could be dramatically different, the static or static-even scheduling type may not achieve the best load balance. We decided to use the *runtime* scheduling for a parallel loop generated by the pre-pass in multithreaded code generation. Therefore, the decision regarding scheduling is deferred until run-time, and a better load balance can be achieved based on the decision made by the OMP_SCHEDULE environment variable and the OpenMP library at run-time.

3.3 Multithreaded Code Generation

The multithreaded code generator consists of many modules such as variable classification, privatization, array lowering, loop analysis, enclosing-while-loop generation for *runtime*, *dynamic* and *guided* scheduling, post-pass *threadprivate* handler and stack optimization. Essentially, it converts the OpenMP constructs to multithreaded code at the IL0 level. See the example in Figure 3. For the *worksharing* loop in the routine *parwork* with the scheduling type *dynamic*, the multithreaded code generation involves: (i) generating a runtime dispatch and initialization (*__kmpc_dispatch_init*) routine call to pass global loop lower-bound, upper-bound, stride, and all other necessary information to the runtime system; (ii) generating an enclosing while loop to dispatch *loop-chunk* at runtime through the *__kmpc_dispatch_next* routine supported in the library; (iii) localizing the loop lower-bound, upper-bound, and privatizing the loop control variable '*k*' and local defined stack variable '*x*'. With the *MET* technology [1], one threaded entry, or T-entry[1], is created within the *parwork*() for the *parallel* regions. The *T-entry* parwork_par_region() corresponds to the semantics of the *parallel* region. The call *__kmpc_fork_call* spawns a team of threads to execute the threaded codes in parallel.

```
void parwork() /* OpenMP C code sample */
{  double a[1000], b=1000;
   int       k;
#pragma omp parallel shared(a, b) private(k)
   {    int x = 7;
#pragma omp for schedule(dynamic)
        for (k=0; k<16; k++) {  x = x + b*b; a[k] = a[k] + b * x;  }
   }
}
entry extern void  _parwork()  /* IL0 pseudo-code after MT-code generation */
{   ... ...
    b = 1000.00 (F64)                    /* F64 denotes the 64-bit float type */
    __kmpc_fork_call(…, __parwork_par_region, &a, &b)
    goto L46
    T-entry __parwork_par_region(ap, bp)
    {  prv_x = 7;   prv_k = 0
       if  (1000 > prv_k) {
           t0 = (* F64)bp;   lower  = 0;   upper  = 999;   stride = 1;
           __kmpc_dispatch_init(..., lower, upper, stride, ...)
        L33:
           t3 = __kmpc_dispatch_next(..,, &lower, &upper, &stride)
           if ((t3 & upper>=lower) != 0(SI32)) {
              prv_k = lower
           L17:
              prv_x = prv_x + t0 * t0
              ((* F64)ap)[prv_k] = ((* F64)ap)[prv_k] + t0 * prv_x
              prv_k = prv_k + 1
              if (upper >= prv_k)    goto L17
              goto L33
           }
       }
        __kmpc_barrier(...)
       T-return
    }
   L46: ... ...
    return
}
```

Figure 3. Pseudo-code After Multithreaded Code Generation

3.4 Aggressive Code Motion

In this Section, we present an optimization -- aggressive code motion that lifts all read-only memory de-references from inside of a region/loop/section to outside of a region/ loop/section. Essentially, the idea is that we do pre-load a memory de-reference into a register temporary right after T-entry, if a memory de-reference can be proved to be a read-

[1] In [1], T-entry refers strictly to the entry point of a threaded region, or T-region, which is the section of code enclosed between a T-entry and its matching T-return. In this paper, we use T-entry to refer to the threaded entry or region, as this use is unambiguous from the context and often interchangeable.

only memory de-reference based on the load-store analysis and memory disambiguation. In Figure 3, for example, the memory de-reference of *bp* is lifted outside the loop and pre-loaded into a register temp *t0*, and the memory de-reference of *bp* is replaced by a load of register temp *t0*.

The benefit of the aggressive code motion is that it reduces the overhead of a memory de-referencing, since the value is preserved in a register temporary for the read operation. The second benefit is that it enables advanced well-known optimizations such as software pipelining, and vectorization if the memory de-references in array subscript expressions are lifted outside the loop. See another example in Figure 4, the address computation of array involves the memory de-references of the member *lower* and *extent* of the dope-vector, the compiler lifts the memory de-references of *lower* and *stride* outside the *m*-loop, because the compiler knows that all references to members of the dope-vector are *read-only* memory references inside the parallel *do* loop.

```
real allocatable:: w(:,:)
… …
!$omp parallel do shared(x), private(m,n)
do  m=1, 1200                !!  Front-End creates a dope-vector for allocatable
   do n=1, 1200              !!  array w
      w(m, n) = …      ➔   dv_baseaddr[m][ n] = …
   end do
end do
… …
T-entry(dv_ptr …)   !! Threaded region after multithreaded code generation
    … …
    t1 = (P32 *)dv_ptr->lower                    !! dv_ptr is a pointer that points
    t2 = (P32 *)dv_ptr->extent                   !! to dope-vector of array w
    do prv_m=lower, upper
       do prv_n =1, 1200                              ! EXPR_lower(w(m,n)) = t1
         (P32 *)dv_baseaddr[prv_m][prv_ n] = …  ! EXPR_stride(w(m,n)) = t2
       end do
   end do
T-return
```

Figure 4. An Example of Aggressive Code Motion

In general, the aggressive code motion enables a number of high-level optimizations such as loop unroll-and-jam, loop tiling, and loop distribution as well. It resulted a very good performance benefit in many real large applications.

3.5 Support Nested Parallelism

Explicitly expressing nested parallelism is supported by the OpenMP specification. However, most of existing OpenMP compilers do not fully support nested parallelism, since the OpenMP-compliant implementation is allowed to serialize nested parallel regions, even when the nested parallelism is enabled by the environment variable OMP_NESTED or routine omp_set_nested(). For instance, the SGI's compiler supports nested parallelism only if the loops are perfectly nested. Given that broad classes of applications, such as imaging processing and audio/video encoding and decoding algorithms, have shown performance benefits by exploiting nested parallelism. We implemented the compiler and the runtime library support needed for full nested parallelism in the Intel compiler. In addition, there are a number of ways to control nested parallelism. For example, the *num_threads* clause can be added to a parallel region *pragma* line to overwrite the number of threads the runtime system will attempt to use for only that region. Note that this setting will not persist to any subsequent or nested parallel regions. In Figure 5, (a) shows a nested parallel region sample code, and (b) shows its corresponding IL0 pesudo-code generated by the C++ compiler Front-End.

(a) A Nested Parallel Region Code Sample

```
#include<omp.h>
void nestedpar()
{   static double a[1000];
    int    id;
#pragma omp parallel private(id)
   {       id = omp_get_thread_num();
#pragma omp parallel
       {
          do_work(a, id, id*100);
       }
   }
}
```

(b) IL0 Pseudo-Code Generated by C++ Front-End

```
entry extern void  _nestedpar()
{
  DIR_PARALLEL QUAL_PRIVATE_VAR(id) QUAL_SHARED_VAR(a)
    t0 = _omp_get_thread_num()
    id = t0
    DIR_PARALLEL QUAL_SHARED_VAR(a) QUAL_SHARED_VAR(id)
      t1 = _do_work(&a, id, id * 100(SI32))
    DIR_END_PARALLEL
  DIR_END_PARALLEL
  return
}
```

(c) IL0 Pseudo Multithreaded Code Generated by Parallelizer

```
entry extern void _nestedpar()
{ ...../* P32 denotes the 32-bit pointer type */
  ___kmpc_fork_call(___nestedpar_par_region0)(P32));
  goto L30
   T-entry void __nestedpar_par_region0()
   {    t0 = _omp_get_thread_num();
        prv_id = t0;
       ___kmpc_fork_call(__nestedpar_par_region1)(P32), &prv_id)
       goto L20;
          T-entry void __nestedpar_par_region1(id_p)
          {    t1 = _do_work( &a, *id_p, *id_p * 100)
               T-return
          }
       L20:
     T-return
   }
  L30:
  return
}
```

Figure 5. An Example of MT-codegen for Nested Par-Regions

As showed in Figure 5 (c), there are two threaded entries, or T-entries, created within the original function nestedpar(). The T-entry *__nestedpar_par_region0*() corresponds to the semantics of the outer *parallel* region, and the T-entry *__nestedpar_par_region1*() corresponds to the semantics of the inner *parallel* region. For the inner *parallel* region in the routine nested*par*, the variable *id* is a *shared* stack variable for the inner *parallel* region. Therefore, it is accessed and shared by the team of threads created for the inner *parallel* region through the T-entry argument *id_p*. Note that the variable *id* is a private variable for the outer *parallel* region, since it is a local defined stack variable.

In addition, as we see in Figure 5 (c), there are no extra arguments on the *T-entry* for sharing local static array '*a*', and there is no pointer de-referencing inside the *T-region* for sharing the local static array '*a*' among all threads in the teams of both the outer and inner *parallel* regions. This uses our compiler technique presented in [2] for sharing static data environment among threads; it is an efficient way to avoid the overhead of argument passing across T-entries.

3.6 Workqueuing Model

The workqueuing model [7] was proposed to exploit task-level or irregular parallelism. This model allows the user to parallelize control structures that are beyond the scope of those supported by the OpenMP programming model, while still fitting into the framework defined by the OpenMP specification. In particular, the workqueuing is a simple and flexible programming model for specifying units of work that are not pre-computed at the start of the worksharing construct. See a simple example in Figure 6.

```
void wq_test(LIST *p)
{
#pragma intel omp parallel taskq shared(p)
  {  while (p != NULL) {
        #pragma intel omp task captureprivate(p)
        {
           wq_workitem(p)
        }
        p= p->next;
     }
  }
}
```

Figure 6. A While-Loop with Workqueuing Pragmas

The *parallel taskq* pragma specifies an environment for the '*while loop'* in which to enqueue the units of work specified by the enclosed *task* pragma. Thus, the *while* loop's control structure and the enqueuing are executed by single thread, while the other threads in the team participate in dequeuing tasks from the *taskq* queue and executing them. The clause *captureprivate* ensures that a private copy of the pointer *p* is captured at the time each task is being enqueued, hence preserving the sequential semantics.

To support the workqueuing model as the Intel OpenMP extension, the Intel C++ compiler's OpenMP support has been extended throughout its various components. First, the IL0 intermediate language has to be expanded to represent the new workqueuing constructs and clauses. The front-end parses the new pragmas and produce IL0 representation of the workqueuing code for the middle-end. The OpenMP parallelizer generates the multithreaded code corresponding to workqueuing constructs. More implementation details of workqueuing model described in paper [1].

4. Multithreaded Runtime Library

The Intel OpenMP runtime library represents a complete redesign at a high level, with only bottom level components re-used from the previous Intel OpenMP runtime library. It remains the backwards compatible in the functionality and performance with the previous Intel runtime library. This section describes some features of the Intel runtime library together with its high level architecture.

4.1 Runtime Library Architecture

The Intel OpenMP runtime library has been designed to exploit nested and sibling parallelism for satisfying the requirements of users using OpenMP in their applications. The typical OpenMP user community has strong roots in scientific high-performance parallel computing. Common uses of the OpenMP in this space are parallelizing entire application executables, with the main thread of control is controlled by the OpenMP programmers. There is also an increasing use of OpenMP mixed with Message Passing Interface* (MPI*) for large problem solving. In addition to the traditional uses identified above, users are starting to use OpenMP in applications where a programmer has little control over the main thread of execution. This scenario is fairly common in applications controlled by Graphical User Interfaces (GUI's), such as those applications built with the Microsoft Foundation Classes (MFC), whereby graphical sub-system controls the main thread of execution and makes calls into the user's application. This programming model is also common whenever the programmer is writing libraries that are called by others – the library writer has little control over the calling environment. Such scenarios often result in multiple system threads invoking the OpenMP – a situation we term *sibling parallelism.*

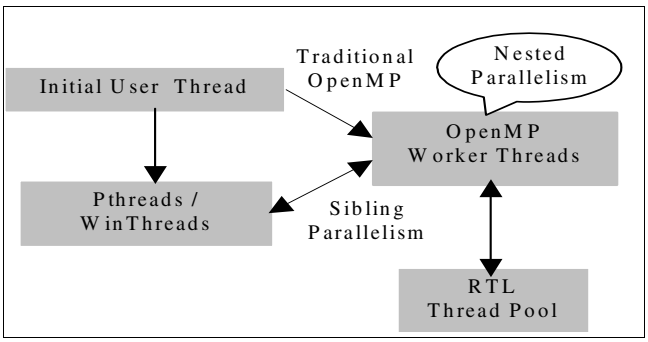

Figure 7. Intel OpenMP Runtime Library Architecture

Figure 7 shows an overview of Intel's OpenMP runtime support. The implementation of the Intel OpenMP runtime library strives to: (a) provide right and rich functionalities, (b) provide good performance, (c) provide good portability and extensibility, (iv) provide hooks to other tools that are part of multithreaded software development. The following subsections describe design considerations and features of the Intel OpenMP runtime library.

4.2 Runtime Support for Nested Parallelism

The specification for OpenMP provides some information on how nested parallelism should be handled. When the Intel runtime library was extended to support the nested

parallelism, it was designed to conform to this specification. The specification supports nested parallelism in OpenMP simply by allowing the use of OpenMP parallel teams nested within already parallel OpenMP regions. By default, the nested parallelism is disabled and nested regions will be serialized, that is they will create a new team containing one thread. This feature must first be enabled either via the environment variable OMP_NESTED or with the routine *omp_set_nested*. There are several methods to control how many threads are used in the various parallel regions:

```
// Start with one thread
omp_set_num_threads(3)
#pragma omp parallel
{    // three threads used here
     omp_set_num_threads(29)
     #pragma omp parallel
     {   // three teams have 29 threads each here  }
     #pragma omp master
          omp_set_num_threads(178);
     #pragma omp parallel
     {    // one team has 178 threads here
          // two teams have 29 threads here
     }
}
#pragma omp parallel
{  // 3 threads here also
   #pragma omp parallel
   {    // 3 teams of 3 threads each here
        // note how the 29 and 178 settings are lost.
   }
}
```

Figure 8. An Example of Configuring the Number of Threads

- An implementation-specific environment variable has been added in the new library. The KMP_MAX_THREADS variable allows the user to set the maximum number of threads the runtime library will use for OpenMP threads. This includes the initial thread, OpenMP worker threads created and being used, OpenMP worker threads waiting in the free pool, and system threads that were created by the user who then subsequently started to exploit sibling parallelism. This allows the user to limit the number of threads to the number of processors, insuring that an application or a library used by an application does not oversubscribe the system with OpenMP threads.
- The OMP_NUM_THREADS environment variable is used to specify default number of threads that the runtime library will try to use whenever creating a new parallel region. Unless users override this setting, the library will attempt to use this many threads at every level, until the KMP_MAX_THREADS limit is reached.
- The routine omp_set_num_threads() is an API call that allows the user to specify how many threads the runtime system should try to use at the next parallel region encountered by the thread that made the call. In traditional one-level fork/join OpenMP it only really makes sense for the original starting thread to make this call. With nested parallelism support, any thread can make this call and teams that thread subsequently creates will be affected by the new setting. This setting is somewhat persistent as shown in Figure 8.
- num_threads(n) is a clause that the user can place on the parallel pragma line. This setting specifies how many threads the runtime system will attempt to use for only that parallel region. This setting is not persistent at all and only applies to its own region.

Figure 8 is an example of how to configure the number of threads through those runtime library calls that control the amount of parallelism for achieving better performance.

4.3 Support Sibling Parallelism

One of desired features beyond the OpenMP model is to enable support for exploiting the sibling parallelism. This model allows different system threads to start the OpenMP teams and vice-versa. In supporting the sibling parallelism, a majority of the work necessary to support the nested parallelism is already required, as presented in previous subsection. The issues we had to address in the design of supporting sibling parallelism in the Intel OpenMP runtime library are centered around the following questions:

- Should those sibling system threads share the OpenMP threadprivate variables?
- Should system threads created from within OpenMP team of threads return the same value for the function call omp_get_thread_num() that their OpenMP creator thread returns? That is, Should the new thread be considered a part of the team that the parent thread belonged to?

```
int pthread_create(userarg,
                   userfunc(), ……)
{
   … ….
 #pragma omp parallel
          num_threads(1) if(false)
   {
        userfunc(userarg);
   }
   … …
   return …;
}
```

```
__crt_init()
{
   int rc;
   … …
#pragma omp parallel
        num_threads(1) if(false)
  {
     rc = main(argc, argv);
  }
   … …
   return rc;
}
```

Figure 9. An Example of Exploiting Sibling Parallelism

Our decisions are settled on not sharing thread identifiers between system threads and their OpenMP parent, and on not sharing threadprivate variables among system threads. System threads are essential flat with respect to each other, just like WinThreads. The primary reasons for this decision were both ease of use for application programmers and ease of conceptual understanding. The conceptual model of sibling parallelism we envision is the following: each system thread created by the system (i.e., not created by OpenMP thread) has an OpenMP parallel region around

with exactly one thread. Programmatically, we represent this with a simple rewrite of two system routines: the __crt_init() routine that calls the users' main(), and the pthread_create() (CreateThread() on Windows) routine that starts system threads on user routines (see Figure 9).

Since each system thread is at the top of its own nested OpenMP hierarchy, it should be noted that a forked system thread would return the *false* to *omp_in_parallel*(), even if created from an OpenMP worker thread. This makes sense, since the new system thread may have no connection to the worker thread that created it, and could have its entry point anywhere doing possibly unrelated work. This allows any model of parallelism and does not force the programmer to make an arbitrary connection between two unrelated threads. If the newly created thread is considered a pseudo-member of its parent's team, then many questions would arise; such as to whether it should participate in barrier pragmas. This would be very difficult since the new thread might have no way of getting to the barrier pragma without a long jump or other contorted method. If the programmer does, however, wish for the newly created thread to share the work of an OpenMP worker thread, it is a simple task to store the result of the omp_get_thread_num() in a private variable that can then be shared by the two threads.

4.4 Runtime Library Performance Tuning

Efficient execution of the OpenMP applications requires the runtime to maintain a thread pool rather than starting and stopping system threads at each parallel region. Therefore, the thread pooling is an essential feature of the runtime. The OpenMP allows orphaned directives that require run-time computation of binding rules to determine how to interpret the directives. Efficient computation of these binding rules is another important feature of the runtime.

Compared to the previous Intel OpenMP runtime library, sibling and nested parallelism require a level of indirection in order to find which sibling or nested team the current thread is a member of. This indirection is a potential source of performance loss. However, we were able to optimize the performance to minimize this penalty. In real, coarse-grained, applications we have observed no performance penalty in going to the new runtime library. In fine-grained micro-benchmarks, the new runtime incurs minimal penalty for most cases.

Another important issue in the design of the OpenMP runtime library centers on the question of what to do with idle threads while they are waiting, whether it be in the thread pool between parallel regions or waiting for a synchronization event. The Intel OpenMP runtime library provides two types of control for this: (i) an environment variable indicating if the user is looking to optimize turnaround time or system throughput because of resource sharing with other jobs or users, (ii) some variables that control the amount of time spinning when idle before falling asleep. The environment variable KMP_LIBRARY can be set to *turnaround* or *throughput*. The default value is "*throughput*" to provide a pretty safe environment whereby creating more threads than processors in compute intensive applications, or accidental sharing of the machine, does not result it terrible performance yields() the processor to other threads or jobs more often than the turnaround library. Both libraries also provide variables to control the amount of time that threads spin at barriers before going to sleep. The environment KMP_BLOCKTIME allows the user to specify about how much time each thread should spend spinning. The user can also adjust this setting at runtime using the kmp_set_blocktime() API call. When adjusted at run-time, the setting applies to the system thread that called it as well as any OpenMP worker threads under it in the nested OpenMP hierarchy. This new setting is especially important for Hyper-Threading (HT) enabled processors. On a HT-enabled processor more than one thread can be executing on the same processor at the same time. This means that both threads have to share that processor's resources. This makes spin-waiting extremely expensive since the thread that is just waiting is now taking valuable processor resources away from the other thread that is doing useful work. Thus, when using Hyper-Threading, the *blocktime* should be very short so that the waiting thread sleeps as soon as possible allowing still useful threads to more fully utilize all processor resources.

5. Performance Results

We have conducted our performance measurements with a set of selected benchmarks to validate the effectiveness of our OpenMP implementation in the Intel high-performance compilers. The multithreaded codes generated by the Intel compiler are highly optimized with architecture-specific, and advanced scalar and array optimizations assisted with our aggressive memory disambiguation. The performance measurement of two micro-benchmarks matrix multiply *matmul* (256x256) and memory copy *memcopy* (4096) is carried out on an Intel Hyper-Threading technology enabled single-processor system running at 2.66GHz, with 2048M memory, 8K L1-Cache, and 512K L2-Cache.

The performance scaling is derived from serial execution (SEQ) with Hyper-Threading technology disabled, and multithreaded execution with one thread and two threads with Hyper-Threading technology enabled. In Figure 3, we show the normalized speed-up of the two chosen micro-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++ and Fortran compiler executing with one thread and two threads, respectively. As shown in Figure 10, the *matmul* (OMP2 w/ HT) achieved a 45% performance improvement by the second threads running on the second logical processor. No multithreading overhead is observed

for one thread run comparing with the serial run. The multithreaded code of the *memcopy* does show a 7% performance degradation due to the overhead of thread creation and forking, synchronization, scheduling at run-time, and memory de-referencing for sharing local stack variables (OMP1 w/ HT), but the second thread running on the second logical processor contributed to the overall 9% performance gain (OMP2 w/ HT).

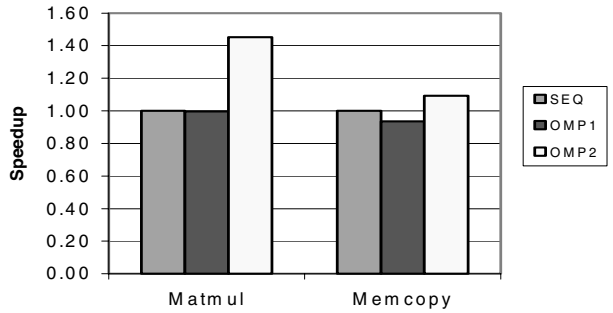

Figure 10. Performance of Two Micro-benchmarks

Figure 11 shows the performance results of three well-known benchmarks: *N-queens* (13x13), *Strassen* (1024×1024 double-precision floating-point matrix), and *Permanent* (11x11 matrix), those benchmarks are written with the Intel workqueuing model [1][7] using *parallel*, *taskq* and *task* pragmas. The performance speedup ranges from 3.44x to 5.22x on an Intel® Xeon™ system with four processors running at 1.6 GHz, with 8K L1 cache, 256K L2 cache, 1MB L3 cache per processor, and 2GB of shared RAM on a 400MHz system bus. The performance measurement has been conducted with both Hyper-Threading Technology disabled and enabled.

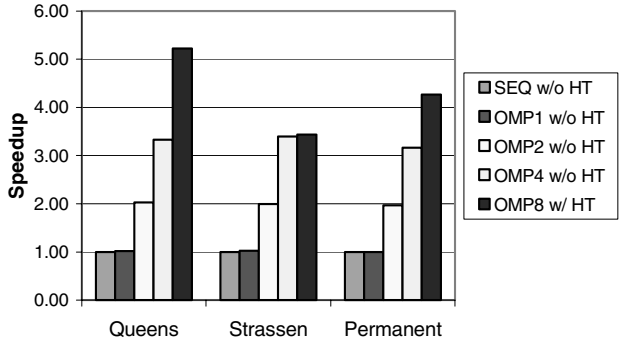

Figure11. Performance of Workqueuing Benchmarks

We disabled Hyper-Threading technology while measuring the performance of serial run of sequential code, and 1-, 2-, and 4-thread run of threaded code generated by the Intel compiler. In this way, we can guarantee that all threads were scheduled on different physical processors, because there is no guarantee that two threads will not be scheduled onto the same physical processor when Hyper-Threading technology is enabled, even though the number of threads is less than the number of physical processors. With Hyper-Threading technology disabled, the speedups of *N-queens*, *Strassen* and *Permanent* benchmark are 3.32, 3.39, and 3.16 respectively, with 4-thread run (OMP4) over the serial run. Note that the runtime overhead of all three threaded codes is very small and not notable. We enabled the Hyper-Threading technology for the 8-thread run (OMP8), the speedup is 5.22 for *N-queens*, 3.44 for *Strassen*, 4.27 for *Permanent*. Thus, the performance gain due to the Hyper-Threading technology is 57% for *N-queens*, 1.5% for *Strassen*, and 35% for *Permanent*. For the *Strassen*, we only saw 1.5% gain from the Hyper-Threading technology, which is mainly limited to memory bandwidth for the given array size of 8MB (more detailed analysis is in the scope of our next paper on performance study).

In addition to the IA-32 performance measurement, in order to evaluate our implementation in the Intel compiler for the OpenMP support on the Intel Itanium Architectures, we conducted the performance measurement with NAS Parallel Benchmarks Suite, which is parallelized with the OpenMP programming model, on a dual-processor Intel Itanium processor-based SMP system running at 800MHz (512K L2 cache, 1MB L3 cache per processor) with 1GB memory. The NAS Parallel Benchmarks is a popular benchmarking suite, written in Fortran 77, which is often used for the performance evaluation on multiprocessor system.

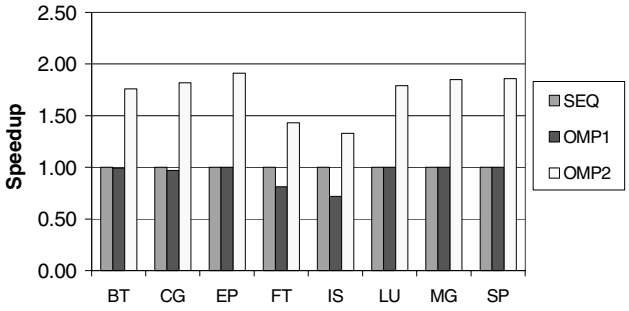

Figure 12. Performance of NAS Parallel Benchmarks

We have been using the Class-A problem sizes for our measurement. The performance improvement of the NAS benchmarks is shown in the Figure 12. The speedups are measured and computed based on the execution time of serial run of each benchmark. The speedup ranges from 1.33 for IS (Integer Sorting) to 1.91 for EP (Embarrassing Parallelism). The concluding remark derived from our results is that the multithreaded code generated by the Intel compiler achieved a good speedup on the dual-processor Itanium SMP system. Note that the overhead of the multithreaded code for BT, CG, EP, LU MG, and SP is not notable with one thread run, however, we saw 19% and 28% performance slowdown for FT and IS, respectively, with multithreaded code running with one thread comparing to their serial code. More detailed performance analysis is in the scope of our next paper on performance study.

6. Conclusion and Future Work

In this paper, we presented the compiler and runtime support of OpenMP in the Intel compiler for the OpenMP directive-guided multithreading. We also demonstrated that performance gains are achieved on Intel platforms based on a set of benchmarks. The Intel OpenMP C++ and Fortran compiler has been designed and implemented to leverage the Pentium and Itanium architecture features. This has been achieved by tightly integrating OpenMP pragma- and directive-guided parallelization with advanced well-known optimizations while generating efficient threaded codes for exploiting parallelism at various levels. The performance results show that OpenMP programs compiled with the Intel C++/Fortran compiler achieved a good performance gain on Intel Hyper-Threading technology enabled Pentium 4 processor-based single- and multi-processor systems, and as well as on Intel Itanium Processor Family (IPF) based multiprocessor systems. One important observation we have is that exploiting thread-level parallelism causes inter-thread interference in caches, and places greater demands on memory system. But, the Hyper-Threading Technology in Intel Pentium 4 processor hides the additional latency, so that there is only a small impact on the whole program performance, hence, we achieved the overall performance gain by exploring the use of logical processor. With Intel's Hyper-Threading and compiler technology, we can shrink the processor-memory performance gap and achieve desired performance gain. In the future, our work is heading in the following directions:

- Investigate the possibility of more aggressive memory optimizations, and identify opportunities of exploiting multi-level parallelism to leverage new architecture and micro-architecture features, and add compiler support of workqueuing model for the Fortran 95 language
- Support *teamprivate* clause that allows the user to specify that what was a threadprivate variable should now be shared among the threads of a new nested team. What was unique to the thread that created the nested parallel region should now be shared among it and its children in the new team
- The usefulness of Intel's KMP_MAX_THREADS extension raises the question of extending the OpenMP standard to include this environment variable. A proposed name is OMP_MAX_THREADS.

Acknowledgements

The authors thank all members of the Intel compiler team for their great work in designing and implementing the Intel C++/Fortran high-performance compiler. In particular, we thank Paul Grey, Aart Bik, Ernesto Su, Hideki Saito, Dale A. Schouten for their contribution in PAROPT projects, Max Domeika and Diana King for the OpenMP C++/C front-end support, Bhanu Shankar and Michael L. Ross for the OpenMP Fortran front-end support, Knud J. Kirkegaard for IPO support, and Zia Ansari for PCG support. Special thanks go to Grant Habb, Bill Margo and the compiler group at KSL for developing and tuning the OpenMP runtime library. We would like to thank the Intel Russia iNNL OpenMP validation team for developing OpenMP test suites and extensive testing of the Intel compiler.

References

[1] Ernesto Su, Xinmin Tian, Milind Girkar, Grant Haab, Sanjiv Shah, Paul Petersen, "Compiler Support for Workqueuing Execution Model for Intel SMP Architectures", *EWOMP 2002 Fourth European Workshop on OpenMP Roma, Italy,* September 18-20th, 2002

[2] Xinmin Tian, Aart Bik, Milind Girkar, Paul Grey, Hideki Saito, Ernesto Su, "Intel® OpenMP* C++/Fortran Compiler for Hyper-Threading Technology: Implementation and Performance*", Intel Technology Journal,* Vol. 6, Q1, 2002 http://www.intel.com/technology/itj

[3] OpenMP Architecture Review Board, "OpenMP C and C++ Application Program Interface," Version 2.0, March 2002, http://www.openmp.org

[4] Aart Bik, Milind Girkar, Paul Grey, and Xinmin Tian, "Automatic Intra-Register Vectorization for the Intel® Architecture," To appear in *International Journal of Parallel Programming*, April 2002.

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

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

[7] Sanjiv Shah, Grant Haab, Paul Petersen, and Joe Throop, "Flexible Control Structures for Parallelism in OpenMP," in *Proceedings of the First European Workshop on OpenMP (EWOMP),*http://www.it.lth.se/ewomp99/papers/grant.pdf

[8] Fred Chow, Sun Chan, Robert Kennedy, Shin-Ming Liu, Raymond Lo, and Peng. Tu, "A new algorithm for partial redundancy elimination based on SSA form," in *Proc. of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation,* June 1997, pp. 273-286.

[9] 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.

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

[11] Debbie Marr, Frank Binns, David L. Hill, Glenn Hinton, David A. Koufaty, J. Alan Miller, and Michael Upton, "Hyper-Threading Technology Microarchitecture and Architecture," *Intel Technology Journal,* Vol. 6, Q1, 2002. http://www.intel.com/technology/itj

---

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

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