

# Compiler Support of the Workqueuing Execution Model for Intel<sup>®</sup> SMP Architectures

Ernesto Su, Xinmin Tian, Milind Girkar  
Intel Compiler Lab, Intel Corporation  
3600 Juliette Lane, Santa Clara, CA 95052  
{ernesto.su, xinmin.tian, milind.girkar}@intel.com

Grant Haab, Sanjiv Shah, Paul Petersen  
KAI Software Lab, Intel Corporation  
1906 Fox Drive, Champaign, IL 61820  
{grant.haab, sanjiv.shah, paul.petersen}@intel.com

## ABSTRACT

*Programs with irregular patterns of dynamic data structures and/or those with complicated control structures such as recursion are hard to parallelize efficiently. The workqueuing model allows the user to exploit irregular parallelism, which is beyond the scope of what is supported by OpenMP<sup>®</sup>. We have integrated this model into the Intel<sup>®</sup> C++ high-performance compiler's OpenMP parallelizer. In this paper, we give an overview of the workqueuing model and describe in detail both the generated multithreaded code as well as the multithreaded run-time library routines supporting it. We also present preliminary performance results of a set of benchmarks and applications measured on Intel SMP architectures.*

## 1 INTRODUCTION

In recent years, OpenMP<sup>®</sup> has gained momentum in both industry and academia to become the de-facto standard parallel programming model for shared-memory machines. However, while OpenMP is fit to parallelize applications with regular parallelism, it has difficulties handling applications with irregular parallelism such as those with recursive function calls or others that operate mainly on list- or tree-based dynamic data structures. To address this issue, the workqueuing model was proposed in [5]. It was designed as a simple extension to the OpenMP C/C++ standard [4] instead of a radically new paradigm, so that it's easy for OpenMP programmers to grasp.

The Intel<sup>®</sup> C++ high-performance compiler fully supports the OpenMP C/C++ standard; details about this compiler and its OpenMP implementation can be found in [6]. This paper focuses on the design and implementation issues in extending the Intel C++ compiler to support the workqueuing model in a unified and consistent manner within its existing OpenMP parallelization framework, thus greatly expanding the range of applications that can be parallelized by using OpenMP-like pragmas. The key point is not the novelty of the workqueuing model, but rather the fact that adding just two new pragmas to OpenMP makes it expressive enough to guide the compiler to generate efficient multithreaded code to exploit the irregular parallelism present in many applications.

This paper is organized as follows. First, we show a high-level overview of the Intel compiler architecture as well as the implementation of its OpenMP support. Next, we present an overview of the workqueuing model, briefly describing the syntax and semantics of the two new pragmas introduced.

Then, we go over the implementation of the workqueuing model, with an emphasis on the integration of this new model into the compiler's existing OpenMP framework, and explain how the generated multithreaded code works. Given the importance of run-time support of the workqueuing execution model, we also present details of its implementation. Finally, we show the performance results of several applications that are very difficult to be parallelized with worksharing constructs, but which can be efficiently parallelized with workqueuing constructs to obtain reasonable speedups on Intel SMP Architectures.

## 2 COMPILER OVERVIEW

In this Section, we give a high-level overview of the Intel high-performance C++/Fortran compiler. As illustrated in Figure 1, 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. This makes it possible that most optimization phases in the compiler, including the OpenMP parallelization phase, be applicable based on the IL0 representation regardless of the source language. Therefore, 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). These optimizations can be classified into:

- Program restructuring, code lowering, and peephole optimizations
- Interprocedural optimizations (IPO) such as inlining and partial inlining, partial dead-code elimination, IP constant propagation, code and data layout
- Parallelizing optimizations: OpenMP directive-guided and automatic parallelization, automatic vectorization
- High-Level Optimizations (HLO) that improve cache behavior and overall memory performance by means of loop transformations (loop unrolling, loop interchange, loop fusion and distribution) and data prefetching
- Scalar optimizations such as copy propagation, partial redundancy elimination (PRE) and partial dead store elimination (PDSE)
- Target-specific optimizations such as store-forwarding elimination, branch prediction, register allocation, advanced instruction selection and scheduling

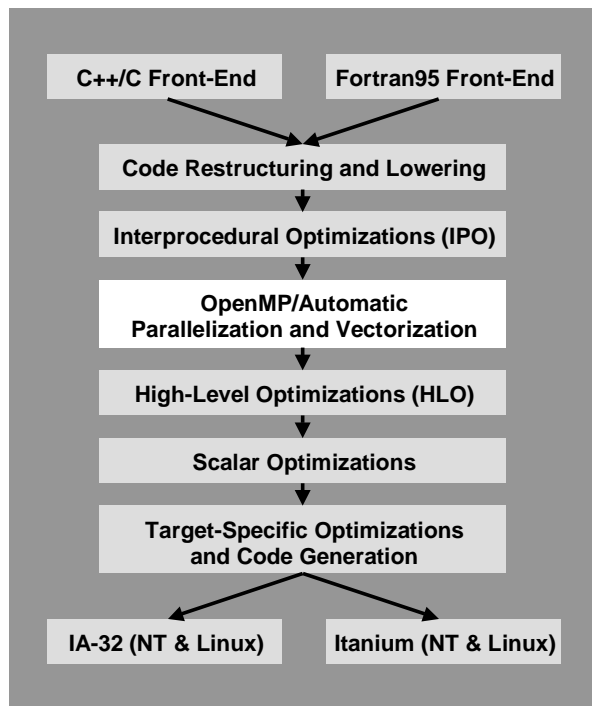


Figure 1: Compiler Architecture Overview

A number of compiler techniques have been developed for parallelization in the Intel compiler [1, 6], the most notable of which is the *Multi-Entry Threading* (MET) [6]. Its basic idea is that the compiler does not have to create a separate routine for each threaded region of code, which would have been required if the outlining technique [2, 3] were used. Instead, it simply generates a threaded entry and a threaded return to enclose the threaded code region corresponding to a given parallel region or loop. The key concept is to keep all newly generated multithreaded code inlined within the original user-defined routine. This greatly simplifies implementation and has other advantages as well [1].

OpenMP parallelization in the Intel compiler's back-end includes many phases and components:

- A pre-pass that transforms OpenMP sections constructs into `do/for` constructs with an appropriate `IF` statement to branch to the correct section of code
- A work-region-node (WRN) graph builder that builds a hierarchical graph to represent the code regions that are enclosed in OpenMP constructs
- A loop analyzer to extract the loop structure and its associated loop control variable, lower and upper bounds, etc.
- An ILO-level code generator for multithreaded code that calls the Intel OpenMP run-time library, *libguide*, developed at Intel KSL
- A variable privatizer that performs privatization of `private`, `firstprivate`, `lastprivate`, and `reduction` variables
- A post-pass that generates code to allocate thread-local storage for `threadprivate` variables.

One of the primary goals of the Intel compiler design is to have OpenMP parallelization be tightly integrated with other advanced optimizations to generate efficient multithreaded code that gains a speed-up over optimized uniprocessor code. Therefore, an effective optimization phase ordering has been devised in the Intel compiler to make sure that all transformations and optimizations enabled before the OpenMP parallelization, including code restructuring, `Igoto` optimizations, IPO inlining and constant propagation, preserve legal OpenMP program semantics and information necessary for parallelization. The phase ordering also ensures that all optimizations after the OpenMP parallelization, such as automatic vectorization, loop transformations, 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.

### 3 WORKQUEUEING MODEL OVERVIEW

#### 3.1 Workqueuing Model Concepts

The workqueuing model allows the user to parallelize control structures that are beyond the scope of those supported by the OpenMP model, while attempting to fit into the framework defined by OpenMP. In particular, the workqueuing model is a flexible mechanism for specifying units of work that are not pre-computed at the start of the worksharing construct. For `single`, `for` and `sections` constructs all work units that can be executed are known at the time the construct begins execution. The workqueuing pragmas `taskq` and `task` relax this restriction by specifying an environment (the `taskq`) and the units of work (the tasks) separately.

The `taskq` pragma specifies the environment within which the enclosed units of work (tasks) are to be executed. From among all the threads that encounter a `taskq` pragma, one is chosen to execute it initially. Conceptually, the `taskq` pragma causes an empty queue to be created by the chosen thread, and then the code inside the `taskq` block is executed single-threaded. All the other threads wait for work to be enqueued on the conceptual queue. The `task` pragma specifies a unit of work, potentially executed by a different thread. When a `task` pragma is encountered lexically within a `taskq` block, the code inside the `task` block is conceptually enqueued on the queue associated with the `taskq`. The conceptual queue is disbanded when all work enqueued on it finishes, and when the end of the `taskq` block is reached.

Many control structures exhibit the pattern of separated work iteration and work creation, and are naturally parallelized with the workqueuing model. Some common cases are C++ iterators, `while` loops, and recursive functions.

If the computation in each iteration of a `while` loop is independent, the entire loop becomes the environment for the `taskq` pragma, and the statements in the body of the `while` loop become the units of work to be specified with the `task` pragma. The conditional in the `while` loop and any

modifications to the control variables are placed outside of the `task` blocks and executed sequentially to enforce the data-dependencies on the control variables. C++ STL iterators are very much like the `while` loops just described, whereby the operations on the data stored in the STL are very distinct from the act of iterating over all the data. If the operations are data-independent, they can be done in parallel as long as the iteration over the work is sequential. This type of `while` loop parallelism is a generalization of the standard OpenMP worksharing for loops. In the worksharing `for` loops, the loop increment operation is the iterator and the body of the loop is the unit of work. However, because the `for` loop iteration variable frequently has a closed form solution, it can be computed in parallel and the sequential step avoided.

Recursive functions also can be used to specify parallel iteration spaces. The mechanism is similar to specifying parallelism via the `sections` pragma, but is much more flexible because it allows arbitrary code to sit between the `taskq` and the `task` pragmas, and because it allows recursive nesting of the function to build a conceptual tree of `taskq` queues. The recursive nesting of the `taskq` pragmas is a conceptual extension of OpenMP worksharing constructs to behave more like nested OpenMP parallel regions. Just like nested parallel regions, each nested workqueuing construct is a new instance and is encountered by exactly one thread. However, the major difference is that nested workqueuing constructs do not cause new threads or teams to be formed, but rather re-use the threads from the team. This permits very easy multi-algorithmic parallelism in dynamic environments, such that the number of threads need not be committed at each level of parallelism, but instead only at the top level. From that point on, if a large amount of work suddenly appears at an inner level, the idle threads from the outer level can assist in getting that work finished. For example, it is very common in server environments to dedicate a thread to handle each incoming request, with a large number of threads awaiting incoming requests. For a particular request, its size may not be obvious at the time the thread begins handling it. If the thread uses nested workqueuing constructs, and the scope of the request becomes large after the inner construct is started, the threads from the outer construct can easily migrate to the inner construct to help finish the request.

Since the workqueuing model is designed to preserve sequential semantics, synchronization is inherent in the semantics of the `taskq` block. There is an implicit team barrier at the completion of the `taskq` block for the threads that encountered the `taskq` construct to ensure that all of the tasks specified inside of the `taskq` block have finished execution. This `taskq` barrier enforces the sequential semantics of the original program. Just like the OpenMP worksharing constructs, it is assumed the user is responsible for ensuring that either no dependences exist or that dependencies are appropriately synchronized between the `task` blocks, or between code in a `task` block and code in the `taskq` block outside of the `task` blocks.

## 3.2 Description of Workqueuing Constructs

The syntax, semantics and allowed clauses are designed to resemble OpenMP worksharing constructs. Most of the clauses allowed on OpenMP worksharing constructs have a reasonable meaning when applied to the workqueuing pragmas. See [4] for more information and background about the semantics of many of the clauses described below.

### 3.2.1 `Taskq` Construct

```
#pragma intel omp taskq [clause[[,clause]. . . ]
    structured-block
```

where *clause* can be any of the following:

```
private( variable-list )
firstprivate( variable-list )
lastprivate( variable-list )
reduction( operator : variable-list )
ordered
nowait
```

**private:** Create a private, default-constructed version for each object in *variable-list* for the `taskq`. Also implies `captureprivate` on each enclosed `task`. The original object referenced by each variable has an indeterminate value upon entry to the construct, must not be modified within the dynamic extent of the construct, and has an indeterminate value upon exit from the construct.

**firstprivate:** Create a private, copy-constructed version for each object in *variable-list* for the `taskq`. Also implies `captureprivate` on each enclosed `task`. The original object referenced by each variable must not be modified within the dynamic extent of the construct and has an indeterminate value upon exit from the construct.

**lastprivate:** Create a private, default-constructed version for each object in *variable-list* for the `taskq`. Also implies `captureprivate` on each enclosed `task`. The original object referenced by each variable has an indeterminate value upon entry to the construct, must not be modified within the dynamic extent of the construct, and is copy-assigned the value of the object from the last enclosed `task` after that `task` completes execution.

**reduction:** Perform a reduction operation with the given *operator* in enclosed `task` constructs for each object in *variable-list*. *operator* and *variable-list* are defined the same as in the OpenMP Specifications [4].

**ordered:** Perform `ordered` constructs in enclosed `task` constructs in original sequential execution order. The `taskq` directive to which the `ordered` is bound must have an `ordered` clause present.

**nowait:** Remove implied barrier at the end of the `taskq`. Threads may exit the `taskq` construct before completing all the `task` constructs queued within it.

### 3.2.2 Task Construct

```
#pragma intel omp task [clause[ , ]clause]... ]
    structured-block
```

where *clause* can be any of the following:

```
private( variable-list )
captureprivate( variable-list )
```

**private:** Create a private, default-constructed version for each object in *variable-list* for the task. The original object referenced by the variable has an indeterminate value upon entry to the construct, must not be modified within the dynamic extent of the construct, and has an indeterminate value upon exit from the construct.

**captureprivate:** Create a private, copy-constructed version for each object in *variable-list* for the task at the time the task is enqueued. The original object referenced by each variable retains its value but must not be modified within the dynamic extent of the task construct.

### 3.2.3 Combined Parallel and Taskq Construct

```
#pragma intel omp parallel taskq \
    [clause[ , ]clause]... ]
    structured-block
```

where *clause* can be any of the following:

```
if( scalar-expression )
num_threads( integer-expression )
copyin( variable-list )
default( shared | none )
shared( variable-list )
private( variable-list )
firstprivate( variable-list )
lastprivate( variable-list )
reduction( operator : variable-list )
ordered
```

Clause descriptions are the same as for the OpenMP parallel construct or the taskq construct above as appropriate.

## 4 IMPLEMENTATION OF THE WORKQUEUEING MODEL

To enable the workqueueing model, the Intel C++ compiler's OpenMP support has been extended throughout its various components. First, the ILO intermediate language has to be expanded to represent the new workqueueing constructs and clauses. Then, the front-end has to parse the new pragmas and produce consistent ILO representation of the workqueueing code for the back-end. In turn, the OpenMP back-end has to generate the code transformations corresponding to the new workqueueing constructs.

An important guiding principle behind the implementation of the workqueueing model in the Intel compiler is that it must be a natural extension to the existing OpenMP framework in the

compiler. Therefore, we have tightly integrated its implementation into the compiler's existing OpenMP parallelization phases described in the previous section, while minimizing the amount of new code needed in the process. For example, by adding just a few lines to the WRN graph builder we have been able to extend it to handle the workqueueing constructs seamlessly. Similarly, we have been able to reuse, with no changes whatsoever, the post-pass that handles threadprivate variables. As a result, it was possible for the workqueueing model's back-end support to be implemented in roughly three weeks by an engineer who has had no previous exposure to this model.

Most of the new code has gone into enabling the ILO-level multithreaded code generator, which lowers workqueueing constructs into equivalent ILO statements and calls to the Intel OpenMP run-time library to carry out the semantics of those constructs. Special attention has been paid to make sure that the same compiler technologies that we have developed before for the OpenMP implementation is equally applicable to the new code that supports the workqueueing model. For instance, the MET technique is especially relevant to this new model due to the natural tendency of workqueueing constructs to be nested more deeply and frequently than worksharing constructs in typical applications.

## 4.1 Multithreaded Code Generation

The best way to explain the code transformations that occur during the compilation of workqueueing constructs is by means of a simple example, such as the while loop shown in Figure 2.

```
1 void test1(LIST p)
2 {
3     while(p != NULL)
4     {
5         do_work1(p);
6         p = p->next;
7     }
8 }
```

Figure 2: A Simple while Loop

This is a natural candidate to be parallelized using the workqueueing model<sup>1</sup>. A programmer can express the parallelism by annotating the loop with a parallel taskq pragma and the work in the loop body with a task pragma, as shown in Figure 3. As explained in Section 3, 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 loop's control structure and the enqueueing are executed single-threaded, while the other threads in the team participate in

<sup>1</sup> Because of its simplicity, this while loop can also be parallelized with an OpenMP single nowait pragma (but would require that all the threads traverse the link list instead of just one). Nevertheless, this simple example is instrumental in helping us illustrate the basic ideas.

dequeuing the work from the `taskq` queue and executing it. The `captureprivate` clause ensures that a private copy of the link pointer `p` is captured at the time each task is being enqueued, hence preserving the sequential semantics.

```

1 void test1(LIST p)
2 {
3   #pragma intel omp parallel taskq shared(p)
4   {
5     while (p != NULL)
6     {
7       #pragma intel omp task \
8         captureprivate(p)
9       {
10        do_work1(p);
11      }
12      p = p->next;
13    }
14  }
15 }

```

Figure 3: Loop with Workqueuing Pragmas

#### 4.1.1 Front-End Transformations

The compiler's front-end generates an ILO representation of the workqueuing code as shown in Figure 4, where the while loop has been lowered into `if` and `goto` statements, and each workqueuing pragma has been converted into an equivalent pair of ILO directive and its matching end directive, which helps the WRN graph builder define the boundaries of the construct.

The compiler's OpenMP back-end, in turn, creates new data structures to handle private and shared variables, and lowers the ILO directives into multithreaded ILO code that explicitly calls various routines in the Intel OpenMP run-time library to enqueue/dequeue the tasks and to manage and synchronize the threads.

```

1 void test1(p)
2 {
3   DIR_PARALLEL_TASKQ SHARED(p)
4   if (p != 0)
5   {
6     L1:
7     DIR_TASK CAPTUREPRIVATE(p)
8     do_work1(p)
9     DIR_END_TASK
10    p = p->next
11    if (p != 0)
12    {
13      goto L1
14    }
15  }
16  DIR_END_PARALLEL_TASKQ
17  return
18 }

```

Figure 4: Front-End Transformations

#### 4.1.2 Run-time Data Structures

The data structures for private and shared variables for the workqueuing model have different requirements from those used for worksharing. Typically, each worksharing thread executes all of the loop iterations scheduled for that thread with just one invocation of the threaded entry, so allocating its private variables as automatic objects on the threaded entry's stack is sufficient to guarantee that the private objects be accessible across iterations. In contrast, in the workqueuing model, the single-thread execution of a `taskq` may need multiple invocations to its threaded entry, because the `taskq` execution is suspended when the queue is full and resumed later, as will be explained in the next subsection. To preserve the values of the private variables of the `taskq` across one invocation of its threaded entry to the next, one cannot allocate private objects as automatic variables on the stack of the `taskq`'s threaded entry. Our approach is essentially to allocate the private objects on the stack of the routine that invokes the `taskq`'s threaded entry. In addition, addresses of shared variables are also stored in a similar fashion so they can be accessed by the `taskq` and its enclosed `taskqs` or `tasks`.

At compile-time, two `struct` types are defined for each `taskq` construct. The first one, `shared_t`, holds the pointers to its shared, `firstprivate`, `lastprivate` and reduction variables. The other `struct`, `thunk_t`, has a pointer to `shared_t`, in addition to fields that hold the private copies of variables listed in the `taskq` as `private`, `firstprivate`, `lastprivate`, `captureprivate`, or `reduction`. At compile-time, a pointer to each `struct` is created, while the actual objects they point to are instantiated at run-time by invoking workqueuing library routines.

For the while loop in this example, we would have the following symbol table entries generated at compile-time:

```

typedef struct shared_t {
... /* fields for internal use */
/* pointers to shared variables below */
p_ptr;
};

auto struct shared_t *shareds;

typedef struct thunk_t {
... /* fields for internal use */
struct shared_t *shr; /* = shareds */
/* private variables below */
p; /* due to captureprivate */
};

auto struct thunk_t *taskq_thunk;

```

where the automatic pointers `shareds` and `taskq_thunk` are allocated outside of the `taskq`'s threaded entry. In addition to the private variables, `thunk_t` holds enough information about the code and data of a `taskq` so that its execution, if suspended due to a full queue, can be resumed at a later time.

### 4.1.3 Back-End Transformations

Figure 5 is an IL0 representation of the parallelized while loop emitted by the OpenMP back-end’s multithreaded code generator. It is interesting to note that, with the exception of the IL0 directives removed by this transformation, all of the original IL0 statements from Figure 4 remain unchanged (except for localized transformations to access private and shared variables) and in the same relative order as in the original routine `test1()`; these statements are highlighted in bold in Figure 5. This is an indication that, throughout the multithreaded code generation, the compiler only inserts new statements but does not have to reorder them or move them to new routines (which would be required if outlining were used), thus simplifying compiler implementation thanks to the MET technique.

Using this technique, three threaded entries, or T-entries<sup>2</sup>, are created within the original function `test1()`: the parallel T-entry `test1_par_taskq()` (lines 6-16 in Figure 5) corresponds to the semantics of the “parallel” portion of the combined `parallel taskq` pragma; the `taskq` T-entry `test1_taskq()` (lines 18-50) corresponds to the “taskq” portion of the combined pragma; and nested within it is the task T-entry `test1_task()` (lines 36-40), which corresponds to the enclosed `task` construct.

In a nutshell, this is what happens at run-time. While every thread executes `test1_par_taskq()`, only one thread proceeds to execute `test1_taskq()`, which is like a skeletal version of the while loop and whose main purpose is to enqueue, on every iteration, the work specified in `test1_task()`. The threads that do not execute `test1_taskq()` become worker threads that dequeue the tasks and execute them. A more detailed description of the multithreaded code in Figure 5 follows.

The Intel OpenMP library routine `__kmpc_fork_call()` invoked in line 3 creates a team of threads. The use of this routine in the worksharing context is explained in [6] and its use here in the workqueuing model is no different. Basically, the parameters to this routine are<sup>3</sup>: the parallel T-entry, and the addresses of shared or other variables that will be accessed in the parallel region.

Thus, every thread calls `test1_par_taskq()`, whose parameter is a pointer to the shared variable `p` (see line 6). All the threads execute this T-entry and attempt to allocate an instance of `thunk_t` and `shared_t` by calling the library routine `__kmpc_taskq()` in line 8, but only one of the threads will succeed in allocating a `thunk_t` for the `taskq_thunk` and will proceed to call `test1_taskq()`

(line 12) with `taskq_thunk` as an argument. The other threads fail the test in line 10 so they skip to line 14 and call `__kmpc_end_taskq()`, which turns them into worker threads.

```

1 void test1(p)
2 {
3   __kmpc_fork_call(test1_par_taskq, &p)
4   goto L3
5
6   T-entry test1_par_taskq(p_ptr)
7   {
8     taskq_thunk = __kmpc_taskq(
9       test1_taskq, 4, 4, &shareds)
10    shareds->p_ptr = p_ptr
11    if (taskq_thunk != 0)
12    {
13      test1_taskq(taskq_thunk)
14    }
15    __kmpc_end_taskq(taskq_thunk)
16    T-return
17  }
18
19  T-entry test1_taskq(taskq_thunk)
20  {
21    if (taskq_thunk->status == 1)
22    {
23      goto L2
24    }
25    if (*(taskq_thunk->shr->p_ptr) != 0)
26    {
27      L1:
28      task_thunk = __kmpc_task_buffer(
29        taskq_thunk, test1_task)
30      task_thunk->p =
31        *(taskq_thunk->shr->p_ptr)
32      if (__kmpc_task(task_thunk) != 0)
33      {
34        __kmpc_taskq_task(taskq_thunk, 1)
35        T-return
36      }
37      goto L2
38
39      T-entry test1_task(task_thunk)
40      {
41        do_work1(task_thunk->p)
42        T-return
43      }
44    }
45    L2:
46    *(taskq_thunk->shr->p_ptr) =
47      (*(taskq_thunk->shr->p_ptr))->next
48    if (*(taskq_thunk->shr->p_ptr) != 0)
49    {
50      goto L1
51    }
52    __kmpc_end_taskq_task(taskq_thunk)
53    T-return
54  }
55  L3:
56  return
57 }

```

Figure 5: Back-End Transformations

Arguments passed to `__kmpc_taskq()` are needed by this routine to allocate memory for `thunk_t` and `shared_t`, and include the size of private variables to be put in `thunk_t` (4 bytes for `p`), the size of pointers added to `shared_t` (4 for `p_ptr`), and the `taskq` T-entry `test1_taskq()`, which is also stored in `taskq_thunk`. To complete the preparation of these data structures, line 9 initializes `shareds->p_ptr` with `&p`.

<sup>2</sup> In [2], 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.

<sup>3</sup> We shall omit library routine parameters that are irrelevant to our discussion. This also applies to other library routines that will be described later in this section

The main purpose of `test1_taskq()` is to enqueue the tasks by running through the loop's control structure. Lines 24 and 43 are the loop test statements, and line 42 is the pointer update statement. Notice the substitution of `p` with the expression `*(taskq_thunk->shr->p_ptr)`, which is necessary to access this shared variable.

On every iteration before actually enqueueing the task, one must first allocate its `thunk_t`, which is pointed to by `task_thunk`. This is done by the call to `__kmpc_task_buffer()` in line 27. An argument to this routine is `taskq_thunk`, based on which `task_thunk` is initialized. Another argument is the task T-entry `test1_task()`, whose address is stored in `task_thunk` for later reference. Furthermore, to satisfy the `captureprivate` semantics, line 28 copies, or "captures," the value of the shared variable `p` into the task's private copy of `p` stored in its `task_thunk`.

The actual enqueueing of a task is done by the library routine `__kmpc_task()` in line 29. This routine extracts the task T-entry `test1_task()` from the `task_thunk` that was passed in as an argument, and enqueues it on the `taskq`'s queue. A return value of zero means that the queue is not full, permitting the `taskq` execution to enqueue the next task.

On the other hand, a non-zero return value indicates that the queue has become full. The execution of the `taskq` is suspended, and will be resumed later by potentially a different thread. To do this, the library routine `__kmpc_taskq_task()` is called in line 31 to enqueue the `taskq_thunk` itself. Later, a worker thread that dequeues it from the queue will resume execution of the `taskq` at the location `L2`: in line 41. To accomplish this, a jump table value (integer 1 in the example) is passed in as an argument to `__kmpc_taskq_task()`. This value is stored in `taskq_thunk` and must uniquely identify this call site because there may be several tasks enclosed in the same `taskq`. This value must also be non-zero because zero is reserved for the first execution of the `taskq`. Based on this value, the jump table at the beginning (lines 20-23) of the `taskq` determines whether the current run of `test1_taskq()` is the first one or the continuation of an earlier run that has been suspended, and accordingly transfers execution to the correct location.

What happens to the thread that enqueues the `taskq_thunk`? After it calls `__kmpc_taskq_task()`, it executes the T-return in line 32 to go back to the caller `test1_par_taskq()`, where it calls the library routine `__kmpc_end_taskq()` in line 14, which turns the calling thread into a worker thread.

## 4.2 Front-End Support

The previous section described the fundamental ideas behind our multithreaded code generation for the workqueuing model. Now we shall use the slightly more involved example

in Figure 6 to expose additional aspects of our front-end support for the workqueuing model.

```

1 int test2(LIST p, int n, int m)
2 {
3   int j;
4   #pragma intel omp parallel default(shared)
5   {
6     int i = n * n;
7     #pragma intel omp taskq lastprivate(j)
8     {
9       int k = 0;
10      while(p != NULL)
11      {
12        #pragma intel omp task \
13          captureprivate(p)
14        {
15          k += i;
16          j = do_work2(p, k);
17        }
18        p = p->next;
19      }
20    }
21    return j;

```

Figure 6: A More Complex Example

One of the problems the front-end has to tackle is to find the implicit attributes of variables that are not explicitly listed in a clause. In this example, variables `j` and `p` are listed as `lastprivate` and `captureprivate`, respectively. But what about the other variables, `i`, `k`, `n`, and `m`?

The first step is to handle the implicit privateness of locally declared variables<sup>4</sup>. Essentially, a locally declared automatic variable is treated as a `private` variable of the OpenMP construct that immediately encloses it lexically. Thus, the front-end adds `k` to the `private` list of the `taskq` construct, and adds `i` to the `private` list of the `parallel` construct.

Next, the front-end has to determine the implied `captureprivate` variables. According to the rules explained in Section 3.2.1, a `private`, `firstprivate`, or `lastprivate` variable in a `taskq` construct implies that the same variable is `captureprivate` in enclosed task constructs. In the example, the `taskq` construct has `j` listed as `lastprivate` and `k` found to be `private` in the previous step, so both are included in the `captureprivate` list of the enclosed `task` construct, in addition to the explicitly listed variable `p`.

Then, it finds the implicit shared variables of the `taskq` (note that the syntax of the `taskq` pragma described in Section 3.2.1 disallows an explicit `shared` clause). These are variables visible outside of the `taskq` construct that are accessed within the construct. In order for them to be accessible inside the `taskq`, these variables have to be

<sup>4</sup> Locally declared automatic variables have been first floated to the routine level (mangled to ensure uniqueness) by an earlier phase in the front-end.

added to the `taskq`'s `shared_t` struct. The variables `p` and `i` are implicit shared variables of the `taskq`.

Finally, the front-end finds the shared variables of the parallel construct to be `n`, `j`, and `p`. Note that it does not include `m` and `k` because they are not accessed within the construct (the variable `k` accessed in this routine is a private copy of either the `taskq` or the `task`). The resulting ILO representation out of the front-end is listed in Figure 7, where the implied clauses and variables are highlighted.

```

1 test2(p, n, m)
2 {
3   DIR_PARALLEL PRIVATE(i) SHARED(n, j, p)
4     i = n * n
5     DIR_TASKQ LASTPRIVATE(j) PRIVATE(k) \
6       SHARED(p, i)
7
8     k = 0
9     if (p != 0)
10    {
11      L1:
12      DIR_TASK CAPTUREPRIVATE(p, j, k)
13      k = k + i
14      j = _do_work2(p, k)
15      DIR_END_TASK
16      p = p->next
17      if (p != 0)
18      {
19        goto L1
20      }
21    }
22  DIR_END_TASKQ
23  DIR_END_PARALLEL
24  return j
25 }

```

Figure 7: Implicit Clauses Found by the Front-End

For reference, we also list the generated multi-threaded code in Figure 8. Structurally, it is very similar to the first example shown in Figure 5, and its lines highlighted in bold also have the same meaning as in Figure 5. The actual new code worth mentioning is in lines 44-47 (shaded). These statements implement the copy-out of the `lastprivate` variable `j`. The macro `IS_LAST` is a bit-mask for the corresponding bit in the variable `task_thunk->flags`. This bit is set by the run-time library in the last iteration of the loop. For completeness, it is relevant to point out that line 25 is where `firstprivate` variables would have been initialized if there were any such variables in the example.

## 5 RUN-TIME SUPPORT

The Intel OpenMP run-time library provides several routines that build and manipulate the data structures to support workqueuing. The functional details of each routine is further described in the subsections that follow.

```

1 int test2(p, n, m)
2 {
3   __kmpc_fork_call(test2_par, &n,&j,&p)
4   goto L3
5
6   T-entry test2_par(n_p, j_p, p_p)
7   {
8     i_prv = (*n_p) * (*n_p)
9     taskq_thunk = __kmpc_taskq(
10      test2_taskq, 12, 12, &shareds)
11     shareds->p_ptr = p_p
12     shareds->i_prv_ptr = &i_prv
13     shareds->j_ptr = j_p
14     if (taskq_thunk != 0)
15     {
16       test2_taskq(taskq_thunk)
17     }
18     __kmpc_end_taskq(taskq_thunk)
19     T-return
20
21   T-entry test2_taskq(taskq_thunk)
22   {
23     if (taskq_thunk->status == 1)
24     {
25       goto L2
26     }
27     /* Initialization of firstprivates */
28     taskq_thunk->k = 0
29     if (*(taskq_thunk->shr->p_ptr) != 0)
30     {
31       L1:
32       task_thunk = __kmpc_task_buffer(
33         taskq_thunk, test2_task)
34       task_thunk->p =
35         *(taskq_thunk->shr->p_ptr)
36       task_thunk->k = taskq_thunk->k
37       task_thunk->j = taskq_thunk->j
38       if (__kmpc_task(task_thunk)!=0)
39       {
40         __kmpc_taskq_task(taskq_thunk,1)
41         T-return
42       }
43       goto L2
44
45     T-entry test2_task(task_thunk)
46     {
47       task_thunk->k = task_thunk->k +
48         *(task_thunk->shr->i_prv_ptr)
49       task_thunk->j = do_work2(
50         task_thunk->p, task_thunk->k)
51       if (task_thunk->flags & IS_LAST)
52       {
53         *(task_thunk->shr->j_ptr) =
54           task_thunk->j
55       }
56       T-return
57     }
58     L2:
59     *(taskq_thunk->shr->p_ptr) =
60       (*(taskq_thunk->shr->p_ptr)->next)
61     if (*(taskq_thunk->shr->p_ptr) != 0)
62     {
63       goto L1
64     }
65     }
66     __kmpc_end_taskq_task(taskq_thunk)
67     T-return
68
69   L3:
70   return (j)
71 }

```

Figure 8: Multithreaded Code Showing `lastprivate`

## 5.1 `__kmpc_taskq()` Routine

The `__kmpc_taskq()` routine is called by all threads as they encounter a `taskq` directive. If the routine is called by threads that are not executing code within a `taskq` construct already, only the master thread of the team creates a task queue data structure initializes it, and adds it as a leaf node in the tree of task queues. The other encountering threads then wait at a barrier inside `__kmpc_taskq()` until the first task is enqueued. If, however, it is called by a thread already executing inside a `taskq` construct, the encountering thread creates a task queue data structure regardless of its status as a master thread or worker thread. Flags indicating whether the `taskq` construct contains an `ordered`, `lastprivate`, or `nowait` clause are passed to `__kmpc_taskq()` in a single integer parameter.

Another set of parameters is used in the construction of a `shared_t` buffer that contains pointers to all the shared variables that may be accessed by the code executed within the `taskq` construct. The necessary size of the buffer is an input parameter to `kmpc_taskq()` and a pointer to the allocated buffer is returned as an output parameter. This `shared_t` buffer is allocated for each calling thread and then initialized by each of the threads as they exit the `__kmpc_taskq()` routine.

One input parameter to this routine is a pointer to the `taskq` T-entry as described in Section 4.1.3. The library uses this pointer, along with an input parameter containing the size of the `private` data associated with the `taskq` construct, the pointer to the allocated `shared_t` buffer, and a pointer to the newly created task queue data structure, to construct and return a pointer to the `taskq_thunk`. This thunk encapsulates all the code and data required to execute the `taskq` routine at a subsequent time.

## 5.2 `__kmpc_end_taskq()` Routine

The `__kmpc_end_taskq()` routine is called by all threads that encounter the `taskq` construct. This routine corresponds syntactically to the end of the `taskq` construct. The thread that executes the `taskq_thunk` calls this routine after finishing the code in that thunk. The other encountering threads call `__kmpc_end_taskq()` after returning from the `__kmpc_taskq()` routine and initializing the `shared_t` buffer. The only other input parameter for this routine is a pointer to the `taskq_thunk`, returned by the associated `__kmpc_taskq()` call, which is used solely to find the corresponding task queue data structures in the tree of task queues. There is no return value for this routine.

Each thread calling this routine executes tasks from the queue until all the tasks have been queued by the thread executing the `taskq_thunk` and dequeued by the worker threads. After the last task is dequeued, the worker threads steal tasks from a child queue, if any exist. If a `nowait` clause is present on the `taskq` directive, the worker threads are then

free to exit this routine and continue executing code subsequent to the `taskq` construct. If not, the threads steal work from any ancestor task queues and their descendants and free finished descendent queues until all descendant queues have been freed. Then, one of the threads frees the queue data structures after the rest exit the `__kmpc_end_taskq()` routine. This process ensures that nothing after the end of the `taskq` construct is executed until all code within the construct has been executed, including the code in the enclosed `task` constructs.

## 5.3 `__kmpc_task_buffer()` Routine

The `__kmpc_task_buffer()` routine allocates a `task_thunk` for a `task` construct and returns it to the calling thread. It is called at the start of each `task` construct encountered by the thread executing the code in the `taskq` construct. Input parameters for this routine are the `taskq_thunk` and a pointer to the `task` T-entry as described in Section 4.1.3. The routine uses the `taskq_thunk` to initialize the allocated `task_thunk` pointers to the `shared_t` buffer and task queue data structures corresponding to the enclosing `taskq` construct. The calling thread initializes the `thunk_captureprivate` variables for the `task` construct after it returns from this routine.

## 5.4 `__kmpc_task()` Routine

After `__kmpc_task_buffer()` allocates the `task_thunk`, the thread executing the `taskq` construct code then calls the `__kmpc_task()` routine to enqueue the task on the task queue specified by the `task` thunk input parameter. If the task queue becomes full after the task is enqueued, the routine returns `TRUE`; otherwise, it returns `FALSE`.

If the task happens to be the first one queued for the `taskq` construct, the worker threads are released from the barrier in the `__kmpc_taskq()` routine.

## 5.5 `__kmpc_taskq_task()` Routine

The `__kmpc_taskq_task()` routine is called by the thread executing the `taskq` construct if `__kmpc_task()` returns that the task queue is full. `__kmpc_taskq_task()` enqueues the `taskq_thunk` in a special slot in the queue and returns. The calling thread is then directed back to the `__kmpc_end_taskq()` routine to help dequeue and execute tasks from the queue. The input parameters are the `taskq_thunk` to be queued and a jump table value that is stored in the thunk data. This jump table value enables the `taskq_thunk` to resume execution later after the point of execution where it called `__kmpc_taskq_task()` in the `taskq` routine.

## 5.6 `__kmpc_end_taskq_task()` Routine

The `__kmpc_end_taskq_task()` routine is called at the end of the `taskq` routine. The `taskq_thunk` input parameter is used to access the queue data structures. The queue flags are marked to indicate that all tasks have been queued, which sets in motion the marking of the last `task_thunk` to implement the `lastprivate` clause semantics.

## 6 PERFORMANCE

### 6.1 Benchmark Programs

We have chosen a set of benchmark programs to validate the feasibility of our implementation of the workqueuing model in the Intel<sup>®</sup> C++ high-performance compiler. These programs represent a wide range of application domains. They are difficult to parallelize with the OpenMP `for` pragma because of the recursive control structures and hierarchical data decomposition that are present in them.

**FFT:** This FFT program came from the Cilk™ version 5.1 distribution from MIT (<http://supertech.lcs.mit.edu/cilk>) and was originally written by Matteo Frigo. It is a highly optimized version of the classical Cooley-Tukey FFT algorithm. It calculates the one-dimensional FFT of a vector of  $n$  complex values, and works by hierarchically decomposing the vector into FFTs of smaller vectors. It is optimized aggressively for sub-vectors whose size is a small power of two. A vector size of  $n=2^{22}$  was used.

**Fibonacci:** This is the smallest of the benchmarks programs used, and computes the  $n^{\text{th}}$  Fibonacci number. It is a highly recursive implementation of the well-known equation:  $\text{Fib}(n) = \text{Fib}(n-1) + \text{Fib}(n-2)$ . For the benchmark,  $n=40$  was used.

**Multisort:** This variation of ordinary mergesort is also included in the Cilk version 5.1 distribution, and was originally written by Matteo Frigo and Andrew Stark. It sorts a random permutation of  $n$  32-bit numbers with a fast parallel sorting algorithm by dividing an array of elements in half, sorting each half recursively, and then merging the sorted halves with a parallel divide-and-conquer method rather than the conventional serial merge. As soon as the workqueuing recursion reaches its lower limit, serial quicksort takes over for the smaller arrays. Furthermore, insertion sort is used for arrays below a threshold of 20 elements to avoid the overhead of quicksort. An array size of  $n=2^{24}$  was used.

**Permanent:** This program computes permanents using the standard expansion-by-minors algorithm that is commonly used to calculate determinants. (For permanents, the minors are always added instead of alternating signs.) It takes roughly  $O(n^n)$  time to calculate the permanent for an  $n \times n$  matrix. The calculation of the permanents for the minors of the matrix can all be done in parallel (using workqueuing tasks) at each level of the recursive expansion. An  $11 \times 11$  matrix was used.

**Pipeline:** Demonstrates the workqueuing implementation of a three stage pipeline. The first stage reads in a line from `stdin`, the second stage inverts the case of the characters, and the final stage writes the case-inverted line to `stdout`. Serializing dependences are that the reads from `stdin` as well as the writes to `stdout` must occur in the correct order. This is enforced by using the OpenMP `ordered` pragma in the workqueuing `task` constructs.

**Queens:** This program, which uses a backtracking search algorithm, was originally written by Keith Randall and included in the Cilk 5.1 distribution. The original  $n$ -queens problem's objective is to find a placement for  $n$  queens on an  $n \times n$  chessboard such that none of the queens attack each other. To provide meaningful performance results, the non-deterministic nature of typical space-search algorithms is avoided by counting the number of possible solutions to the problem instead of stopping at the first solution found. We used a board size of  $13 \times 13$ .

**Quicksort:** This is an implementation of the quicksort algorithm found in many algorithms textbooks. While the sorting is done in parallel with the workqueuing model, the merge step is done serially. The program sorted an array with  $10^6$  elements in our experiments.

**Strassen:** Strassen's algorithm for multiplication of large dense matrices use hierarchical decomposition of matrix multiplication by dividing each dimension of the matrix into two sections of equal size. For  $n \times n$  matrices it achieves a run-time complexity of  $O(n^{2.807})$ , which is an improvement over the standard matrix multiplication with a complexity of  $O(n^3)$ . The original program was written by John Larson in Fortran. It was then translated into C and parallelized using workqueuing pragmas; the details of the parallelization can be found in [5]. The matrices used in our experiments had  $1024 \times 1024$  double-precision floating-point numbers each.

### 6.2 Methodology

The benchmark programs were parallelized by using workqueuing pragmas, and their serial versions used in the experiment were obtained by removing such pragmas. Both the parallelized and the serial versions of these programs were compiled with the Intel C++ high-performance compiler, version 7.0 beta. The parallelized programs were compiled with the `-Qopenmp -O2` switches, while the serial ones with the `-O2` switch.

Because the main purpose of these performance experiments is to demonstrate the feasibility of our implementation of the workqueuing model, no significant effort was spent to fine-tune the granularity of these benchmarks to optimize their performance on the target architectures. Clearly, should any of these applications be used in a performance-critical environment, further adjustments to the task granularity may achieve additional performance gains.

Two target platforms were used to execute the benchmarks. They both run Microsoft Windows\* 2000 server operating system, and have the following configurations:

**Platform 1:** The first machine used in our experiments has four Intel® Xeon™ processors running at 550 MHz, 1MB of L2 cache per processor, and 1GB of shared RAM sitting on a 133MHz system bus.

**Platform 2:** The second machine has four Intel® Xeon™ processors running at 1.6 GHz, 256K on-chip L2 cache, 1MB L3 cache per processor, and 2 GB of shared RAM on a 400MHz system bus. The Hyper-Threading Technology feature on these processors has been disabled so that the additional performance benefits it provides do not make our results more complex to analyze.

### 6.3 Results

For every benchmark on each platform, we collected the performance results by timing its execution for the serial version as well as for the parallel version using from one to four threads. Each such run was done three times and the median was used to compute the multithreaded speedup, which is the ratio of the serial execution time divided by the multithreaded execution time. These speedups were then shown in Table 1 and 2 for the two target platforms, respectively.

**Table 1: Performance Results on Intel® Xeon™ 550 MHz**

Benchmarks	Multithreaded Speedup			
	1 Thread	2 Threads	3 Threads	4 Threads
FFT	1.01	1.46	1.75	2.14
Fibonacci	0.99	1.89	2.88	3.66
Multisort	0.92	1.41	1.61	1.74
Permanent	1.00	1.99	3.00	3.98
Pipeline	1.00	1.93	2.72	2.72
Queens	0.99	1.98	2.97	3.95
Quicksort	0.97	1.64	2.04	2.41
Strassen	1.01	1.90	2.63	3.19

**Table 2: Performance Results on Intel® Xeon™ 1.6GHz**

Benchmarks	Multithreaded Speedup			
	1 Thread	2 Threads	3 Threads	4 Threads
FFT	1.04	1.83	2.40	3.10
Fibonacci	1.00	1.92	2.99	3.79
Multisort	1.01	1.89	2.52	2.96
Permanent	1.00	1.97	2.95	3.93
Pipeline	1.00	1.95	2.76	2.76
Queens	1.01	2.03	3.04	4.05
Quicksort	0.96	1.69	2.21	2.92
Strassen	1.02	1.91	2.64	3.39

The column labeled “1 Thread” therefore shows a measure of the overhead of the multithreaded code. Most of these programs exhibit very small overheads. In fact, the sorting benchmarks (Multisort and Quicksort) are the only ones with overheads greater than 1%, the largest of which is 8% seen with Multisort on platform 1. Two benchmarks, FFT and Strassen, show small gains in their 1-thread execution over the serial execution. These are likely due to unexpected changes in cache behavior introduced by the threading code and should be safe to ignore.

Three of the benchmarks show very good scalability (speedup > 3.5 on four threads) across both platforms. They are Fibonacci, Permanent, and Queens. It is likely that the granularity and task queue depth used in the parallelization of these programs happen to be near-optimal for the architectures on which they were run (as mentioned before, none of the benchmarks were intentionally tuned for these platforms).

While Strassen’s scalability is about the same on either platform, both sorting algorithms scale significantly better on platform 2 than on platform 1. This is especially notable with Multisort, whose 4-thread speedup drops from 2.96 on platform 2 to only 1.74 on platform 1. Sorting algorithms are typically memory bound, and these two are no exceptions. The arrays sorted are too large to fit in the cache, so the superior memory bandwidth of Platform 2 greatly benefits their multithreaded performance. Theoretically, if memory bandwidth is not an issue, Multisort should scale slightly better than Quicksort because the its merge stage is also done in parallel. Empirically, this is shown in Table 2, where the 4-thread speedup of Multisort is 2.96, which is slightly higher than Quicksort’s 2.92. FFT is another benchmark that scales much better on platform 2; it is likely that this program is also memory bound.

Finally, it is interesting to mention that Pipeline’s speedup tops out for three threads regardless of the target platform. This is expected because the pipeline used in the experiment only has three stages, so additional threads do not help.

### 7 CONCLUSIONS AND FUTURE WORK

The workqueuing model is a simple yet powerful extension to OpenMP and greatly increases the range of applications that can be parallelized with OpenMP. We have presented in this paper an implementation of the workqueuing model in an existing, commercially available compiler. We have also shown that such an implementation can be done as a natural extension to an existing OpenMP compiler framework. Moreover, we demonstrated empirically that, using the workqueuing model it is easy to obtain good multithreaded performance for applications that are out of the scope of standard OpenMP due to their complicated control structures and dynamic data decompositions.

There are two main aspects to our future efforts with the workqueuing model. The first aims at extending its reach, while the second aims at extending its functionality. For the first goal, we intend to implement the workqueuing model on

more compilers. Logically, the next compiler to have this extension will be the Intel Fortran compiler. For the second goal, we will conduct more research and experiments with the workqueuing model to add features to this model that make it more versatile and yet easier to use. For example, we could add an `if` clause to the `taskq` pragma to control recursion depth for the tree of queues. If the `if` clause's expression evaluates to false, the `taskq` and all enclosed `taskqs` are executed serially. We could call an API function such as `omp_get_taskq_depth()` to calculate the current `taskq` nesting level. Similarly, we could call `omp_get_taskq_count()` to obtain the total number of queues in the current tree. These calls can be used to construct logical expressions for the `if` clause. For example:

```
#pragma intel omp taskq \  
    if (omp_get_taskq_depth() < depth_limit)
```

## 8 ACKNOWLEDGMENTS

The authors wish to thank members of the Intel compiler's front-end team for their constant support. In particular, we wish to thank Diana King for implementing most of the necessary changes in the C++ front-end to support the new workqueuing pragmas, and to thank Clark Nelson for his expert interpretation of the OpenMP standards, which has helped the implementation immensely in times of confusion.

## 9 REFERENCES

- [1] 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*, <http://intel.com/technology/itj/q12001>, Q1 2001
- [2] 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.
- [3] 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.
- [4] OpenMP Architecture Review Board, "OpenMP C and C++ Application Program Interface," Version 2.0, March 2002, <http://www.openmp.org>.
- [5] 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>
- [6] Xinmin Tian, Aart Bik, Milind Girkar, Paul Grey, Hideki Saito, and Ernesto Su, "Intel® OpenMP C++/Fortran Compiler for Hyper-Threading Technology: Implementation and Performance", *Intel Technology Journal*, Vol. 6, Issue 1, 2002, [http://www.intel.com/technology/itj/2002/volume06issue01/art04\\_fortrancompiler/p01\\_abstract.htm](http://www.intel.com/technology/itj/2002/volume06issue01/art04_fortrancompiler/p01_abstract.htm).