

Flexible Control Structures for Parallelism in OpenMP

Sanjiv Shah, Grant Haab, Paul Petersen, & Joe Throop
Kuck & Associates, Incorporated
1906 Fox Drive
Champaign, IL 61820
<http://www.kai.com>
{sanjiv,grant,petersen,jthroop}@kai.com

Abstract

OpenMP cannot handle some very common programming idioms like recursive control and list or tree data structures. We present the Workqueuing model and show it as a natural, flexible, and easy to use extension of OpenMP that is available in a commercially shipping product. A detailed description of Workqueuing is presented, together with performance results and pointers to the source code used.

1 Introduction

Over the last decade, scientists, engineers, and independent software vendors have used directive based parallel programming models to parallelize applications on shared memory parallel (SMP) computers. In the last two years, this programming experience has been codified, and an industry sponsored consortium, OpenMP, has adopted new specifications. The OpenMP Fortran API [OMPF] and the OpenMP C/C++ API [OMPC] were announced at Supercomputing 97 and Supercomputing 98, respectively.

The practical success of the directive based programming model has largely been due to its applicability to array based Fortran applications. In this case, the identification of the computationally intensive loops has been straightforward, and in many important cases, scalable multiprocessor performance improvement is readily obtained.

The use of directive based parallelism for C/C++ and non-array based Fortran codes has been much less wide spread. Although important C/C++ applications have been parallelized by this method, these applications are typically array based, and many of these cases are essentially translation of Fortran array based algorithms into C/C++.

The OpenMP parallel programming model is based on array based computation. Its evident success in the computing community can be attributed to a number of simple factors: it is simple and easy to use; it preserves the original sequential program; and it can be programmed incrementally. Although the OpenMP model satisfies the needs of many array based applications, it is generally understood that many programs exist in C/C++ which do not fit the underlying array based model, such as those with list or

tree structured data or recursive control structures.

We present here simple extensions to the OpenMP model to support C/C++ and Fortran codes that have non-array based data structures or recursive control structures. We show that these extensions are consistent with the OpenMP language and enable programmers to parallelize, without destroying the original sequential program, a large class of programs that previously required vast amounts of restructuring.

Many proposals for parallel C/C++ extensions have been discussed and even implemented by the research community. Our goal has been to engineer an implementation consistent with OpenMP using this body of knowledge and provide this software to developers and researchers in a supported, commercial software package. Kuck & Associates has presented the workqueuing model to the OpenMP working group as an extension to the OpenMP specifications.

In this paper, we review the two basic OpenMP worksharing constructs, the OpenMP `for` pragma and the OpenMP `sections` pragma, and illustrate the difficulty of programming important cases with these constructs. We then show that the addition of two new pragmas, one to generate queues of independent tasks and another to encapsulate independent tasks, solves these problems. We call this addition to OpenMP the workqueuing model. Finally, experimental measurements of parallel speedup are presented for several benchmark programs on three different hardware platforms.

1.1 Limitations of OpenMP

OpenMP currently supports two kinds of worksharing constructs to speed up parallel programs: the OpenMP `for` pragma and the OpenMP `sections` pragma. Unfortunately, neither of these constructs can be utilized to parallelize access to list or tree structured data or to fully parallelize codes with recursive control structures.

1.1.1 The OpenMP `for` Pragma

The OpenMP `for` pragma is the source of parallelism in most OpenMP programs and provides potentially unlimited parallelism. It indicates that iterations of the `for` loop

```
#pragma omp for
for( int i=0; i<n; i++ ) process( data[i] );
```

Figure 1: The OpenMP for Pragma

```
nodeptr list, p;
...
for( p=list; p!=NULL; p=p->next )
    process(p->data);
```

Figure 2: Simple Pointer Chasing Loop

```
nodeptr list, p;
...
// Count the number of iterations
int n=0;
for( p=list; p!=NULL; p=p->next ) n++;

// compute the data pointer for each iteration
nodeptr* q = new nodeptr[ n ];
int i=0;
for( p=list; p!=NULL; p=p->next ) q[i++] = p;

#pragma omp parallel for
for( int i=0; i<n; i++ ) process( q[i] );
```

Figure 3: Parallelization with OpenMP for Pragma

can be run on different threads and provides up to n -way parallelism for an n -iteration loop. (See figure 1 for an example using OpenMP for.)

The limitations of this pragma become obvious when you consider the requirements on the for loop: The number of iterations of the for loop must be computable on entry to the loop. An additional restriction is that all iterations must be more or less independent, (i.e., the data they operate on must be accessible as a function of the loop index). This restriction effectively rules out all while loops and linked data structures for which the only way to precompute the number of iterations is to walk the entire structure, and the only way to access the data for each iteration independently is to allocate an auxiliary data structure.

For example, the simple pointer-chasing loop in figure 2 can be parallelized with OpenMP constructs as shown in figure 3. The first loop in figure 3 can be eliminated by modifying the linked list data structure to keep a count of the number of elements in the list, but the second loop as well as the new and delete cannot be eliminated. There exist many other ways to parallelize the loop in figure 2 using OpenMP constructs just as there should be in any general purpose parallel language, but not one method is concise, scalable and portable. The OpenMP APIs simply were not designed to parallelize such loops.

```
#pragma omp sections
{
    #pragma omp section
    process( a );
    #pragma omp section
    process( b );
}
```

Figure 4: The OpenMP sections Pragma

1.1.2 The OpenMP sections Pragma

The OpenMP sections pragma is used in conjunction with the OpenMP section pragma to indicate static regions of computation that can be simultaneously executed on different threads. The section pragma specifies each such region of computation, such that the use of n section pragmas enables n -way parallelism.

One of the requirements for the section pragma is that it appear immediately inside the sections pragma. This requirement limits the parallelism to be static in nature, in that the parallelism limit is the same as the number of static section pragmas appearing in the code, which is two in figure 4. This parallelism limit in turn limits parallel scalability.

1.2 The Workqueuing Model

This paper proposes a model of explicit parallelization, called the workqueuing model, that addresses the limitations of the OpenMP APIs as discussed in section 1.1.

The basic syntax of the workqueuing model pragmas is as follows:

```
#pragma omp taskq [ clauses ]
#pragma omp task [ clauses ]
```

Logically, a taskq pragma causes an empty queue of tasks to be created. The code inside a taskq block is executed single-threaded. Any task pragmas encountered while executing a taskq block specify that the enclosed work is associated with the queue and can be assigned to any thread, and the unit of work is logically enqueued for subsequent execution.

Taskq pragmas may be nested within either a taskq block or a task block, and in both cases a subordinate queue is formed. The queues logically form an n -ary tree that mirrors the dynamic nesting relationships of the taskq pragmas.

Figure 5 illustrates how workqueuing overcomes the OpenMP limitations exposed in figures 2 and 3 by utilizing the workqueuing taskq and task pragmas to achieve dynamic task creation. Since there is no restriction on the type of code that can appear inside a taskq block or a task block, the workqueuing constructs do not suffer from the for and sections pragma limitations. Assuming the list in figure 5 has n elements, $2n$ -way parallelism is feasible. The flexibility of creating tasks anywhere within the taskq block

```

nodeptr list;
...
#pragma omp parallel taskq
for( nodeptr p=list; p!=NULL; p=p->next ) {
    #pragma omp task
    process( p->data );
    #pragma omp task
    process( p->more_data );
}

```

Figure 5: Pointer Chasing Parallelization with Workqueuing

makes workqueuing a powerful, general-purpose model for parallelism.

2 Workqueuing Target Programs

Most modern programs utilize complex data structures that are not limited to arrays. Computations involving lists and tree-based data structures usually occur within `while` loops and recursive functions. It should not be necessary to introduce auxiliary structures to parallelize such applications. To demonstrate the effectiveness of the workqueuing model, we show how it can be used to parallelize efficiently hierarchical linked lists and recursive control and data structures.

2.1 Hierarchical Linked Lists

When hierarchical data structures are present, parallelism may exist at multiple levels of the hierarchy. Parallelization at different levels may be desirable depending on the tasks being performed. Figure 6 demonstrates how workqueuing can be used to create tasks dynamically at run time. In contrast, the number of OpenMP `section` pragmas appearing in the `sections` block determines a statically specified number of tasks. The corresponding sequential program for the code in Figure 6 is obtained by simply disregarding the pragmas entirely.

2.2 Recursive Control/Data Structures

When presented with tree structured data, the computation usually involves recursive routines. A model that claims to enable parallelization of general purpose programs must be able to represent parallelism in recursive routines concisely. In Figure 7, the processing that happens on each node is independent and can be executed in parallel.

In figure 8, each level of the recursion creates a queue and starts a task to process that node while recursing to the left subchild and then the right subchild. The processing of the data in each node occurs in parallel, with no ordering constraints amongst nodes.

3 Related Work

```

forestptr forests, f;
treeptr t;
branchptr b;
...
#pragma omp taskq
for( f=forests; f!=NULL; f=f->next ) {
    if( destroying_rainforests ) {
        #pragma omp task
        destroy_forest( f );
    } else for( t=f->trees; t!=NULL; t=t->next ) {
        if( logging_trees ) {
            #pragma omp task
            log_tree( t );
        } else for( b=t->branches; b!=NULL; b=b->next ) {
            if( trimming_branches ) {
                #pragma omp task
                trim_branch( b );
            }
        }
    }
}
}
}

```

Figure 6: Hierarchical Linked List Code Example

```

void traverse( Node& node ) {
    process( node.data );

    if ( node.has_left )
        traverse( node.left );
    if ( node.has_right )
        traverse( node.right );
}

```

Figure 7: Sequential Tree Traversal Code

```

void traverse( Node& node ) {
    #pragma omp taskq
    {
        #pragma omp task
        process( node.data );

        if ( node.has_left )
            traverse( node.left );
        if ( node.has_right )
            traverse( node.right );
    }
}

```

Figure 8: Workqueuing Parallelized Traversal Code

OpenMP, X3H5, and PCF Work on the PCF specification by the Parallel Computing Forum led to the standardization effort of the ANSI X3H5 Committee [PCF] [X3H5]. Although the ANSI X3H5 Committee did not produce a final specification, it laid the ground work for the current OpenMP APIs [OMPF, OMPC]. Workqueuing extends the OpenMP API by targeting a larger class of applications than any of the previous attempts.

POSIX Threads POSIX threads offer an API for an extremely general purpose model of parallelism [PThreads]. Just like general purpose assembly language, it can represent all forms of shared memory parallelism including the programs that workqueuing targets.

The representation of parallelism with POSIX threads, however, is not as concise as directive/pragma-based parallel models. Programs written for POSIX threads often have no sequential counterpart, and the principles of sequential programming that all programmers are familiar with no longer apply. This model often results in bugs that are difficult to detect and programs that are difficult to maintain. Workqueuing is a more concise model that also preserves the sequential program semantics.

Cilk Although the workqueuing model was designed independently of Cilk, the two models show some similarities including abstractions like queues of tasks and workstealing [Cilk]. A major design difference is that Cilk uses unstructured control flow, while workqueuing is designed to have a precise block structure. Workqueuing allows the delineation of any block of code as a task using the `task` pragma, while Cilk requires that a task be defined by routine boundaries. Finally, unstructured synchronization (`sync` constructs) may cause divergence from sequential semantics.

Lambda Blocks in Lisp In the workqueuing model, the creation of a task that can be migrated to any thread for execution requires that the `task` construct's execution environment be encapsulated. This notion is similar to the lambda block that most lisp variants employ [LISP].

Futures Futures allow specification of independent work, the status of which is checked sometime into the future [Futures]. Futures are unstructured constructs whereas the workqueuing model has a very definite block structure.

Compositional C++ Compositional C++ (CC++) is a task parallel dialect of C++, which allows the user to create both structured and unstructured task graphs [CC++]. The major enhancement of workqueuing over CC++ is the introduction of flexible control flow. CC++ provides two task parallel control constructs, the `par` block and the `parfor` block. The `par` block executes each statement in the block in a separate task. The `parfor` block executes each

iteration of the following for loop in a separate task. In contrast, the `taskq/task` pair in workqueuing allows the user to parallelize a much greater range of control-flow structures.

Sthreads Sthreads (Structured Threads) is a library and pragma-based notation for parallel programming using the C/C++ programming language [Sthreads]. The Sthreads approach shows some similarities to OpenMP and to the nesting features of the workqueuing model. Sthreads lacks the concept of a parallel region like OpenMP has, so redundant execution of code is not possible. Sthreads offers constructs for a fork-join model, but the dynamic creation of tasks is missing. These factors combine to make workqueuing more general and flexible than Sthreads.

4 Workqueuing Model Concepts

The workqueue programming model is designed to be similar to the OpenMP model. As a result, major benefits of the model are that a programmer can easily write parallel programs that are consistent with the sequential code. In this case consistency implies the following:

- Parallel programs that get the same answers (within roundoff tolerance) as the sequential programs.
- Parallel programs that are identical to the sequential program when the pragmas are ignored.

These same benefits that most attract most software vendors to OpenMP, and should attract them to workqueuing also.

4.1 Workqueuing Pragmas

The workqueue programming model introduces two new pragmas to OpenMP: `taskq` and `task`. Semantically, workqueuing is very similar to OpenMP's worksharing. The major difference is that `taskq` constructs may be nested inside each other. Besides this difference, the binding and nesting rules of worksharing constructs apply to workqueuing constructs.

To take the analogy one step further, the workqueuing `taskq` pragma is akin to the OpenMP `sections` pragma, and the workqueuing `task` pragma is akin to the OpenMP `section` pragma. Furthermore, like the OpenMP `section` and `sections` pragmas, the `task` pragma must be lexically enclosed by the `taskq` pragma. The difference, however, is that `task` can appear anywhere inside `taskq`, whereas `section` must be at the top level inside `sections`.

Logically, a `taskq` pragma causes an empty queue (of tasks) to be created. The code inside a `taskq` block is executed single-threaded. Any `task` pragmas encountered while executing the `taskq` block specify that the enclosed unit of work is associated with the enclosing `taskq` block and can be executed by any thread. This unit of work is

logically enqueued on the queue created by the enclosing `taskq` block and is logically dequeued and executed by any thread.

`Taskq` pragmas encountered when executing either a `taskq` block or a `task` block form a subordinate queue. The whole structure of queues resembles a logical tree of queues, where the root of the tree corresponds to the outermost `taskq` block, and the internal nodes are `taskq` blocks encountered dynamically inside a `taskq` or `task` block. This nesting flexibility is explored further in section 6.

4.2 Workqueuing Synchronization

Synchronization constructs for workqueuing behave similarly to OpenMP's synchronization constructs for worksharing. In fact, the OpenMP `critical` pragma and the OpenMP lock run-time library work exactly the same inside workqueuing constructs.

As is the case for the OpenMP worksharing constructs, the `taskq` construct contains an implicit, but optional, barrier at the end of its block. Inclusion of the `nowait` clause on the `taskq` pragma implies that any thread may proceed past the end of the `taskq` block before all the work associated with that `taskq` has been completed; conversely, the exclusion of a `nowait` clause prevents the threads from thus proceeding. Because of the nesting ability of `taskq` constructs, this implied barrier at the end of the `taskq` block applies only to the threads that encountered a particular instance of the `taskq` construct.

An `ordered` construct is executed in the order that the enclosing `task` was logically enqueued, relative to the other tasks that bind to the same `taskq` construct. This order is the same as the original sequential order of execution. The `taskq` pragma associated with the enclosed `task` construct must include the `ordered` clause. An `ordered` pragma must be enclosed in a `task` block if it occurs anywhere inside a `taskq` block.

Section 6 describes the syntax and semantics of workqueuing constructs in more detail, together with the allowed clauses.

5 Simple Workqueuing Examples

The basic description in section 4 provides enough context to study simple workqueuing examples. Nested `taskq` examples are examined here, since non-nested examples (like the one presented in section 2.1) are typically straightforward.

The theme of the examples in this section is parallelization of operations on tree structured data. Figure 8 in section 2.2 shows the workqueuing parallelization of tree structured data such that all the nodes can be processed in parallel.

At first glance, it appears as if there is little parallelism in the example, because only one task is enqueued on each task queue. Immediately after the first task is enqueued, however, the left subchild is visited, and the pro-

cessing for that node is enqueued on a new queue. In the meantime, one of the worker threads dequeues the task from the top level task queue and executes it.

In this way, a tree of queues mirroring the original data structure is built, and the task at each node of the tree is dequeued and executed by a thread. Practically, the entire data structure should not be mirrored with a tree of task queues; instead, only enough task queues to yield sufficient parallelism should be created at the top levels of the tree.

The workqueuing model presented here could be extended to accommodate limiting the number of task queues in the tree by any of the following methods:

- An `if` clause on the `taskq` pragma could be added. A true value means that the tasks in that task queue are executed in parallel. A false value indicates that the tasks in that task queue are executed sequentially by a single thread.
- A `qdepth(d)` clause on the `taskq` pragma could be added. If the depth of the particular task queue is less than *d*, the tasks enqueued on that task queue are executed in parallel. Otherwise, the tasks are executed sequentially by a single thread.
- A `maxqs(n)` clause on the `taskq` pragma could be added. If fewer than *n* task queues exist in the tree when a particular `taskq` pragma is first encountered, the tasks enqueued on that task queue are executed on multiple threads. Otherwise, the tasks are executed sequentially by a single thread.

In the current implementation of the workqueuing model, a particular method of limiting the task queue tree depth has not yet been chosen because the alternatives warrant further study. Currently, `if` statements around the entire `taskq` block are employed to limit the tree depth.

The examples in this section do not try to limit the depth of the task queue tree to keep them simple. Most practical applications would require a depth limiting mechanism to obtain scalable performance.

5.1 Parallel Preorder Tree Traversal

The preorder traversal in figure 9 assumes that the binary tree computation has a dependence that requires processing a parent node before either child can be processed. The preorder traversal restrictions are honored because the `process` function call must complete before the left and right subtree tasks are queued.

At first glance, maximum parallelism may seem to be limited to two for the preorder traversal code. But as the recursion of the preorder routine is expanded during the tree walk, a tree of task queues is built with two tasks in each queue. The maximum parallelism is only limited by the number of leaves in the tree.

The situations that would prevent good parallel speedup in this case are a highly imbalanced load such that

```

void preorder( Node& node ) {
    #pragma omp taskq
    {
        process( node.data );

        if ( node.has_left )
            #pragma omp task
            preorder( node.left );
        if ( node.has_right )
            #pragma omp task
            preorder( node.right );
    }
}

```

Figure 9: Preorder Traversal of Binary Tree

```

void postorder( Node& node ) {
    #pragma omp taskq
    {
        if ( node.has_left )
            #pragma omp task
            postorder( node.left );
        if ( node.has_right )
            #pragma omp task
            postorder( node.right );
    }
    process( node.data );
}

```

Figure 10: Postorder Traversal of Binary Tree

processing the top node takes much more time than processing the rest of the tree and a tree that doesn't have enough nodes to keep all of the available threads busy.

5.2 Parallel Postorder Tree Traversal

The postorder traversal code in figure 10 assumes that the binary tree has a dependence that requires processing a parent node after both children have been processed. The implicit barrier at the end of the `taskq` construct ensures that the children have been processed before the current node is processed. The parallelism constraints are similar to those discussed in section 5.2.

5.3 Parallel Ordered Tree Traversal

The code in figure 11 executes the `process` function on the left and the current node of the tree in parallel, but executes the `process_in_order` function on each node of the tree in the original sequential order. This feat is accomplished by wrapping the entire processing of the left and right subtrees in an `ordered` construct and by wrapping only the necessary part of the current node in an `ordered` construct.

The processing of `node.data` for the entire tree can be visualized as a left to right wavefront, where the wavefront

```

void ordered( Node& node ) {
    #pragma omp taskq ordered
    {
        if ( node.has_left )
            #pragma omp task
            #pragma omp ordered
            ordered( node.left );

        #pragma omp task
        {
            process( node.data );
            #pragma omp ordered
            process_in_order( node.inorder_data );
        }

        if ( node.has_right )
            #pragma omp task
            #pragma omp ordered
            ordered( node.right );
    }
}

```

Figure 11: Inorder Processing of Some Data

is parallel to the left edge of the tree.

6 Workqueuing Model Details

6.1 Syntax of Workqueuing Pragmas

The syntax 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.

```
#pragma omp taskq [ clause [ clause ] ... ]
```

where *clause* can be any of the following:

`private(list)` – create a private, default-constructed version of the *list* objects for each task.

`firstprivate(list)` – create a private, copy-constructed version of the *list* objects for each task.

`lastprivate(list)` – create a private, default-constructed version of the *list* objects for each task, with object in outer scope copy-assigned from the object in the last enqueued task.

`reduction(operator: list)` – perform a reduction operation in enclosed tasks on the *list* objects with the given *operator*. *Operator* and *list* are defined the same as in the OpenMP APIs.

`ordered` – perform `ordered` constructs in enclosed tasks in original sequential order.

`nowait` – remove implied barrier at the end of the `taskq`.

```
#pragma omp task [ clause [ clause ] ... ]
```

where *clause* can be any of the following:

`private(list)` – create a private, default-constructed version of the *list* objects for this task.

`firstprivate(list)` – create a private, copy-constructed version of the *list* objects for this task.

Notice that a `private` and a `firstprivate` clause are permitted on the `task` pragma, whereas the similar `section` does not permit either. There are two primary reasons for this: Some users of OpenMP have found the absence of these clauses on the `section` pragma lacking, and deferred execution of a task requires encapsulation of its execution environment. These clauses permit this encapsulation to be performed efficiently. Encapsulation is discussed in more detail in the section 6.2.2.

6.2 Workqueuing Pragma Semantics

6.2.1 Taskq Pragma Semantics

The description of `taskq` constructs is broken into two parts, one for the normal case where they resemble work-sharing constructs and one for the nested case where `taskq` constructs are encountered inside `taskq` or `task` constructs.

Normal case: multiple threads encounter `taskq`

A `taskq` construct encountered inside a parallel region creates a logical queue of tasks on which the work represented by enclosed `task` constructs is enqueued. Such a `taskq` construct is executed single threaded by one of the threads that encounters it. Other threads that encounter the construct wait to dequeue and execute work from the queue.

Nested case: single thread encounters `taskq`

A `taskq` construct encountered inside another `taskq` or `task` construct also creates a logical queue of tasks on which the units of work represented by enclosed `task` constructs are enqueued. This new queue is a child of its outer queue. The hierarchical relationship of queues built in this nested way forms a tree of queues, which mirrors the hierarchical control and/or data structures of the program. A small difference between the nested and the normal case is that each thread that encounters a nested `taskq` construct executes it, whereas only one thread that encounters a normal (non-nested) `taskq` construct executes it.

Objects on the `private`, `firstprivate`, and `reduction` lists of the `taskq` pragma have their normal meaning, as listed in section 6.1. The `ordered` clause on the `taskq` pragma indicates that enclosed `task` constructs contain `ordered` constructs that must be executed in the order in which the tasks were enqueued (i.e., the order is the same as in the original sequential program).

The absence of a `nowait` clause means that no thread may proceed past the end of the construct until the current `taskq` construct and all the enclosed `task` and `taskq` constructs have finished. Notice that this says nothing about previously encountered `taskq` constructs — they may still be unfinished. To guarantee that `taskq` constructs A, B, and C and their associated tasks have completed before point X in a program, simply enclose A, B, and C in a `taskq` construct that ends before point X.

The presence of a `nowait` clause means that a thread may proceed past the end of the `taskq` construct once all enclosed `task` constructs have been dequeued.

Objects listed in the `lastprivate` clause of the `taskq` pragma have the original global object copy-assigned from the object in the `task` that was enqueued last. Note that the `lastprivate` clause is for communication of a last value from a `task` to the context outside the `taskq`; it is not intended for any other communication. This clause is very similar to the `lastprivate` clause on an OpenMP `for` pragma or `sections` pragma. In both of the cases above, the body of the `taskq` construct, outside any enclosed `task` or `taskq` constructs, is executed single-threaded, but not necessarily by the same thread during the entire lifetime of the construct.

Requiring a `taskq` construct to be executed by a single thread during its entire lifetime imposes performance limitations and/or large memory requirements on the model. For example, if a `taskq` generates a large number of tasks, the memory needed to store all the tasks is very large. If, on the other hand, a fixed size queue is used, a performance problem surfaces: The `taskq` construct has to wait for the queue to have an available slot before generating more tasks. If the thread executing the `taskq` construct participates in the execution of tasks to alleviate this performance problem, it may end up with a particularly long task, which could lead to starvation of the other threads.

For these reasons, our current workqueuing implementation permits multiple threads to execute the `taskq` construct, as long as only one of them executes it at any single time. However, we permit the thread switch only at enclosed `task` construct boundaries.

Because of the possibility that a `taskq` construct may be executed by different threads, referencing an OpenMP `threadprivate` variable in this section of code may lead to unexpected values. Similar surprises will occur if a `threadprivate` variable is referenced in a `taskq` construct as well as an enclosed `task` construct, because both may be executed by the same thread. For these reasons, `threadprivate` variables may not be referenced in the `taskq` construct outside any `task` constructs.

6.2.2 Task Pragma Semantics

`Task` constructs encountered inside a `taskq` construct are enqueued on the corresponding logical queue. Each `task` construct represents a unit of work in the OpenMP sense and is executed by a single thread.

Data Environment Encapsulation The enqueueing of tasks on a logical queue and their deferred execution while the `taskq` construct generates more tasks requires encapsulation of the task’s data environment, akin to what happens with the OpenMP `for` pragma, where the value of the loop index is encapsulated for each iteration of the `for` loop.

Variables that are private to the `taskq` have their initial values captured for each enqueued `task`. Notice this means for n tasks and p threads, there are $n + 1$ copies of `taskq` private variables, though not all are necessarily live at the same time.

Work Stealing The workqueuing model is designed to permit any thread to execute any task on any queue, which is referred to as work stealing. Work stealing permits all the threads started by the runtime system to stay busy even when the tasks generated by the particular `taskq` that they are executing have finished. In a system that permits dynamic creation and nesting of parallelism, the varying amounts of dynamic parallelism available in different parts of the program and at different levels of nesting make work stealing very important.

In the workqueuing model, work stealing may occur in the nested `taskq` case where a thread executing a particular `taskq` construct runs out of tasks to execute in that queue but finds other tasks ready to execute in other queues (from other parts of the `taskq` tree). Work stealing may also occur in the non-nested case due to the presence of the `nowait` clause.

It should be noted that the queues in the model are merely logical entities. An implementation is free to do away with all the queues, as long as it can satisfy the requirements of all the clauses on the workqueuing pragmas.

7 Workqueuing Application

In this section, the background information and parallelization details for Strassen’s matrix multiply illustrate the application of the workqueuing model.

7.1 Strassen’s Matrix Multiply

Strassen’s algorithm is well known among computational scientists as an algorithm for fast multiplication of large, dense matrices [FisPro]. For square matrices of size $n \times n$, Strassen’s algorithm achieves a run time complexity of $\Theta(n^{\lg 7}) = O(n^{2.807})$ [CLR]. The more familiar matrix multiplication, which we shall refer to as standard matrix multiplication here, has a runtime complexity of $O(n^3)$.

All variants of Strassen’s algorithm use hierarchical decomposition of matrix multiplication by dividing each dimension of the matrix into two equal sized sections.

The matrix multiplication

$$A[1 : 2n][1 : 2n] \times B[1 : 2n][1 : 2n] \Rightarrow C[1 : 2n][1 : 2n]$$

may be decomposed as follows:

$$\left(\begin{array}{c|c} A_{1,1} & A_{1,2} \\ \hline A_{2,1} & A_{2,2} \end{array} \right) \times \left(\begin{array}{c|c} B_{1,1} & B_{1,2} \\ \hline B_{2,1} & B_{2,2} \end{array} \right) \Rightarrow \left(\begin{array}{c|c} C_{1,1} & C_{1,2} \\ \hline C_{2,1} & C_{2,2} \end{array} \right)$$

such that

$$\begin{aligned} \mathcal{M}_{1,1} &\equiv \mathcal{M}[1 : n][1 : n] & \mathcal{M}_{1,2} &\equiv \mathcal{M}[1 : n][n + 1 : 2n] \\ \mathcal{M}_{2,1} &\equiv \mathcal{M}[n + 1 : 2n][1 : n] & \mathcal{M}_{2,2} &\equiv \mathcal{M}[n + 1 : 2n][n + 1 : 2n]. \end{aligned}$$

Standard matrix multiplication is equivalent to the following:

$$\begin{aligned} C_{1,1} &= A_{1,1} \times B_{1,1} + A_{1,2} \times B_{2,1} & C_{1,2} &= A_{1,1} \times B_{1,2} + A_{1,2} \times B_{2,2} \\ C_{2,1} &= A_{2,1} \times B_{1,1} + A_{2,2} \times B_{2,1} & C_{2,2} &= A_{2,1} \times B_{1,2} + A_{2,2} \times B_{2,2}. \end{aligned}$$

Strassen’s algorithm needs only seven submatrix multiplications instead of eight:

$$\begin{aligned} P_1 &= (A_{1,1} + A_{2,2}) \times (B_{1,1} + B_{2,2}) \\ P_2 &= (A_{2,1} + A_{2,2}) \times B_{1,1} \\ P_3 &= A_{1,1} \times (B_{1,2} - B_{2,2}) \\ P_4 &= A_{2,2} \times (B_{2,1} - B_{1,1}) \\ P_5 &= (A_{1,1} + A_{1,2}) \times B_{2,2} \\ P_6 &= (A_{2,1} - A_{1,1}) \times (B_{1,1} + B_{1,2}) \\ P_7 &= (A_{1,2} - A_{2,2}) \times (B_{2,1} + B_{2,2}) \end{aligned}$$

$$\begin{aligned} C_{1,1} &= P_1 + P_4 - P_5 + P_7 & C_{1,2} &= P_3 + P_5 \\ C_{2,1} &= P_2 + P_4 & C_{2,2} &= P_1 + P_3 - P_2 + P_6. \end{aligned}$$

Each of these seven submatrix multiplications is then itself decomposed and the algorithm is applied recursively.

Since Strassen’s algorithm uses more submatrix additions and subtractions than the standard matrix multiplication algorithm, it becomes more expensive after a few steps of hierarchical decomposition when the matrices become small. (The exact crossover point is dependent on the hardware and software implementation and must be chosen experimentally.) Below the performance crossover point in the recursion, standard matrix multiply is employed without further decomposition.

7.2 Parallelizing Strassen’s Algorithm

The hierarchical decomposition aspect of the multiplication is straightforward to parallelize via the workqueuing model. Figure 12 below shows pseudocode for a parallel version of the recursive Strassen’s multiply routine, `strassen_mul`. It takes two square matrices as input, `A` and `B`, and produces a result square matrix, `C`. `r` is the current recursion depth, `pd` is the maximum recursion depth for parallel decomposition, and `t` is the matrix size threshold below which Strassen’s algorithm becomes more computationally expensive than the standard multiplication algorithm.

Note that only the intermediate `p` matrices are computed in parallel. The computation of the final `C` submatrices involves only matrix additions and subtractions and takes negligible time compared to the multiplications performed when computing the `p` matrices.

The desired parallelism depth, `pd`, is calculated by taking into consideration the number of tasks at each level of the recursive parallel decomposition as well as the number of available processors on the target machine. In general, the total number of tasks at the bottom level of the recursion should be a large enough multiple of the number of processors to insure adequate load balancing. Choosing a very large value for `pd`, however, may cause parallel speedup to decrease if the tasks become small relative to the overhead of workqueuing. Experimentation with a particular implementation is necessary to achieve the optimal value.

```

strassen_mul( A[1:2n][1:2n], B[1:2n][1:2n], C[1:2n][1:2n], r ) {
  if ( n < t )
    standard_mul( A[1:2n][1:2n], B[1:2n][1:2n], C[1:2n][1:2n] );
  else {
    allocate p[1:7][1:n][1:n];
    #pragma omp taskq if ( r <= pd )
    {
      #pragma omp task
      strassen_mul( A[1:n][1:n] + A[n+1:2n][n+1:2n],
                    B[1:n][1:n] + B[n+1:2n][n+1:2n],
                    p[1][1:n][1:n], r+1 );

      #pragma omp task
      strassen_mul( A[n+1:2n][1:n] + A[n+1:2n][n+1:2n],
                    B[1:n][1:n],
                    p[2][1:n][1:n], r+1 );

      #pragma omp task
      strassen_mul( A[1:n][1:n],
                    B[1:n][n+1:2n] - B[n+1:2n][n+1:2n],
                    p[3][1:n][1:n], r+1 );

      #pragma omp task
      strassen_mul( A[n+1:2n][n+1:2n],
                    B[n+1:2n][1:n] + B[1:n][1:n],
                    p[4][1:n][1:n], r+1 );

      #pragma omp task
      strassen_mul( A[1:n][1:n] + A[1:n][n+1:2n],
                    B[n+1:2n][n+1:2n],
                    p[5][1:n][1:n], r+1 );

      #pragma omp task
      strassen_mul( A[n+1:2n][1:n] - A[1:n][1:n],
                    B[1:n][1:n] + B[1:n][n+1:2n],
                    p[6][1:n][1:n], r+1 );

      #pragma omp task
      strassen_mul( A[1:n][n+1:2n] - A[n+1:2n][n+1:2n],
                    B[n+1:2n][1:n] + B[n+1:2n][n+1:2n],
                    p[7][1:n][1:n], r+1 );
    }
    C[1:n][1:n] = p[1] + p[4] - p[5] + p[7];
    C[1:n][n+1:2n] = p[3] + p[5];
    C[n+1:2n][1:n] = p[2] + p[4];
    C[n+1:2n][n+1:2n] = p[1] + p[3] - p[2] + p[6];
  }
}

```

Figure 12: Parallelized Strassen’s Algorithm

8 Workqueuing Performance

To evaluate the performance of the workqueuing model, parallel performance measurements were taken using four benchmark programs on three different SMP workstations. In this section, we describe the benchmark programs, the experimental methodology, and give parallel speedup results.

8.1 Benchmark Programs

The benchmark programs used for the performance evaluation were chosen because they are difficult to parallelize using traditional OpenMP techniques, and because they represent a variety of application domains. This difficulty is usually caused by recursive control structures and hierarchical data decomposition which is often difficult to parallelize with the OpenMP `for` pragma. The source code for all of the programs used in these experiments can be obtained from our FTP site at the following URL: ftp://ftp.kai.com/public/workqueuing_examples. Three of them originated from the Cilk distribution from MIT and were parallelized using the workqueuing pragmas. Unless otherwise specified, the serial version of each program used for experiments was obtained by simply

removing the pragmas from the parallel version.

Strassen’s Matrix Multiply Strassen’s algorithm for multiplication of dense matrices and its workqueuing parallelization are described in detail in sections 7.1 and 7.2. John Larson wrote the original Fortran version of this code which was then translated to C. The program used in the performance studies multiplies two $n \times n$ square matrices of double precision floating-point numbers using hierarchical decomposition of the matrix as shown in figure 12.

Fast Fourier Transform This program, originally written by Matteo Frigo, is a highly optimized version of the classical Cooley-Tukey Fast Fourier Transform algorithm [CoTu]. It calculates the one dimensional FFT of a vector of n complex values and is included in the Cilk version 5.1 distribution. The FFT works by hierarchically decomposing the vector into FFTs of smaller vectors and is highly optimized for subvectors for which the size is a small power of two.

Queens The objective of the original N-queens problem is to place n queens on an $n \times n$ chess board in such a way that none of the queens can attack each other. To remove the scheduling indeterminacy in the original program and provide meaningful performance results, the code was rewritten to find and count the number of possible solutions to the problem instead of quitting when any single solution is found. Queens uses a backtracking search algorithm and was originally written by Keith Randall and is included in the Cilk version 5.1 distribution.

Multisort Multisort is a variation of ordinary mergesort [AkSa]. It implements fast parallel sorting by dividing an array of elements in half, sorting each half recursively, and then merging the sorted halves using a divide and conquer approach instead of the usual serial merge. Originally written by Matteo Frigo and Andrew Stark, this program sorts a random permutation of n 32-bit numbers and is included in the Cilk version 5.1 distribution. As soon as the workqueuing recursion reaches its lower limit, serial quicksort is used to sort the smaller arrays. Below a threshold of twenty array elements, insertion sort is employed to avoid the larger overhead of quicksort.

8.2 Experimental Methodology

The benchmark programs were all compiled using the Guide C/C++ component of the KAP/Pro Toolset version 3.7 with default optimization (-O). Two executable versions of each program were created, one with the workqueuing pragmas enabled (parallel) and the other with the pragmas disabled (sequential). We ran each of these programs on three dedicated SMP machines and measured the core algorithm completion time only, since input generation times are significant for some programs.

<i>Program</i>	<i>Platform</i>	<i>Size</i>	T_S	$\frac{T_S}{T_1}$	$\frac{T_S}{T_2}$	$\frac{T_S}{T_3}$	$\frac{T_S}{T_4}$
Strassen	IBM	1536 ²	59.2	1.02	1.97	2.88	3.71
	Compaq	1024 ²	26.5	1.01	1.99	2.92	3.78
	SGI	1280 ²	41.2	1.00	1.93	2.83	3.66
FFT	IBM	2 ²²	20.2	0.99	1.89	2.40	3.26
	Compaq	2 ²³	23.0	1.00	1.71	2.21	2.86
	SGI	2 ²³	80.7	1.02	1.85	2.41	2.87
Queens	IBM	13	55.5	1.05	2.09	3.13	4.16
	Compaq	13	60.2	1.03	2.05	3.07	4.09
	SGI	13	87.0	0.98	1.94	2.91	3.86
Multisort	IBM	2 ²⁴	14.1	0.97	1.87	2.62	3.32
	Compaq	2 ²⁴	14.0	0.95	1.54	1.88	2.44
	SGI	2 ²⁴	18.9	0.97	1.82	2.56	3.31

Table 1: Workqueuing Performance Results

Three different target platforms were used in these experiments: an IBM F50 Model 7025 with four 333 Mhz 604e processors and 1 GB memory running AIX 4.3.2, a Compaq AlphaServer 4100 with four 467 MHz EV56 Alpha processors and 1 GB memory running Tru64 Unix version 4.0D, and an SGI Origin 200 with four 180 MHz R10000 processors and 512 MB memory running IRIX 6.5.

8.3 Performance Results

Performance results of the experiments are shown in table 8. The *Platform* column indicates which of the three SMP machines was used to obtain the corresponding row of performance results, while *Size* denotes the size in elements of the input to the program used on that platform. T_S is the time, in seconds, for the sequential program to run, and T_1 through T_4 designate completion time of the workqueuing parallelized program run using one through four threads, respectively. The columns marked T_S/T_1 through T_S/T_4 provide measurements of speedup, which is the run time of the parallel code using one through four threads relative to the run time of the sequential code.

The T_S/T_1 column indicates that Multisort is a bit slower than the quicksort algorithm used for the sequential time. The speedups in this column indicate memory system effects that are difficult to predict and analyze.

Both Strassen and Queens exhibit near linear or superlinear parallel speedup and, hence, good scalability. Memory bandwidth limitations prevent Multisort from achieving near linear speedup since the ratio of computation to memory access is relatively small when compared to some of the other programs in this group. This effect seems especially severe on the Compaq platform that relies on a deep memory hierarchy to mask the larger gap between memory access latency and processor speed. FFT achieves reasonably good speedup on four processors, but the reasons for sublinear speedup are not yet understood.

9 Conclusions

In this paper we have shown that the workqueuing model forms a simple, natural extension to OpenMP that greatly increases the class of algorithms that can be simply and effectively parallelized. In addition, the examination of the benchmark programs' parallel speedup has shown that it is easy to obtain good parallel performance on algorithms representative of important calculations not easily parallelized with OpenMP.

By the addition of the workqueuing model to the OpenMP C/C++ and Fortran APIs, the domain of codes amenable to effective SMP parallelism would be greatly increased.

The computing community is naturally reluctant to invest major programming efforts in proprietary solutions. Our goal is to encourage developers by making the workqueuing extensions to OpenMP available as part of a commercially supported software package. Kuck & Associates has also proposed the workqueuing model as an extension to the OpenMP APIs, and has thus made it available for adoption and implementation by all vendors.

References

- [FisPro] P. C. Fisher & R. L. Probert, "Efficient Procedures for Using Matrix Algorithms," Automata, Languages, and Programming #14 in Lecture Notes in Computer Science, pp. 413-427, Springer Verlag, 1974.
- [CLR] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, pp. 739-745, McGraw Hill, 1990.
- [OMPC] OpenMP Architecture Review Board, *OpenMP C and C++ Application Program Interface*, Version 1.0, October 1998, <http://www.openmp.org>.
- [OMPF] OpenMP Architecture Review Board, *OpenMP Fortran Application Program Interface*, Version 1.0, October 1997, <http://www.openmp.org>.
- [PCF] Parallel Computing Forum, *PCF Fortran*, Version 3.1, August 1, 1990.
- [X3H5] American National Standards Institute, *Parallel Processing Model for High Level Programming Languages*, ANSI Document Number: X3H5/94-SD2 Revision L, Accredited Standards Committee X3, April 5, 1994.
- [LISP] J. McCarthy, P. W. Abrahams, D. J. Edwards, T. P. Hart, and M. I. Levin, *Lisp 1.5 Programmer's Manual*, 2nd Ed., MIT Press, Cambridge, Mass, 1965.
- [CC++] P. A. Carlin, K. M. Chandy, and C. Kesselman, "The Compositional C++ Language Definition," Technical Report C8-TR-92-02, Computer Science Department, California Institute of Technology, 1992.

- [PThreads] *POSIX System Application Programming Interface: Threads Extension [C Language]*, POSIX 1003.4a, Draft 8, IEEE Standards Department.
- [Sthreads] John Thornley, K. Mani Chandy, and Hiroshi Ishii, "A System for Structured High-Performance Multithreaded Programming in Windows NT," 2nd USENIX Windows Symposium, pp. 67-76, Seattle, Washington, August 3-5, 1998.
- [Cilk] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. "Cilk: An efficient multithreaded runtime system," In Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), pp. 207-216, Santa Barbara, California, July 1995.
- [Futures] Robert H. Halstead, Jr., "Multilisp: A language for concurrent symbolic computation," *ACM Transactions on Programming Languages and Systems*, 7(4):501-538, October 1985.
- [CoTu] James W. Cooley and John W Tukey, "An algorithm for the machine calculation of complex Fourier series," *Mathematics of Computation*, 19(90):297-301, April 1965.
- [AkSa] Selim G. Akl and Nicola Santoro. "Optimal parallel merging and sorting without memory conflicts," *IEEE Transactions on Computers*, C-36(11), November, 1987.