OpenMP Fork/Join Optimization

Princeton Ferro
University of Rochester

Jeff Sandoval
Cray Inc.

Background

OpenMP is a programming model and standard for writing parallel code in C, C++, and Fortran. For example, in C, if one wanted to execute a multithreaded loop:

```
#include <stdio.h>
#include <omp.h>
int main(void) {
    #pragma omp parallel for
    for (int i = 0; i < 10; i++)
        printf("thread %d has %d\n", omp_get_thread_num(),
i);
}
```

The compiler is responsible for translating OpenMP directives (`#pragma omp ...`) into calls into the OpenMP runtime, which is also responsible for supporting a programmer-visible API (`omp_*` functions).

With the above example, the `clang` compiler will outline the body of the for-loop (the parallel region) and translate the pragma into a runtime-specific fork call that will run the outlined function on each available thread (pseudo-IR):

```
define void omp_outlined(%thread_state) {
    // ... iterate over my partition
}
define void main() {
    call @__kmpc_fork_call(@.omp_outlined.)
}
```

Now consider the following code:

```
#include <stdio.h>
int main(void) {
    #pragma omp parallel
    {
        printf("hello (outside)\n");
        #pragma omp single
        {
            #pragma omp task
            printf("Hello 1\n");
            #pragma omp task
            printf("Hello 2\n");
        }
    }
}
```

In OpenMP, one can use `#pragma omp task` to specify a piece of work. Tasks can be executed in any order and other threads may execute tasks when they are idle.

Problem

Cray's implementation of OpenMP can optimize thread joining when it knows that tasks are not present in a given parallel region. However, it is up to the compiler to recognize these cases and pass this information to the runtime. What complicates things further is that tasks don't have to appear lexically nested in a parallel region. We can have a call to a function that invokes a task, for example. The first step is understanding when we can apply our optimization:

A parallel region/procedure P is "task-free" iff:

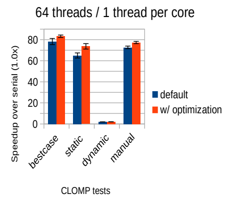
1. It contains no tasks
2. Every non-parallel procedure reachable from P is task-free. (Since parallel regions are invoked on a separate thread, it is okay for them to contain tasks.)

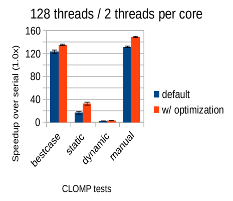
(1) means that there is a list of runtime calls that cannot appear in the procedure, and (2) means that these runtime calls are not reachable in the control-flow from P.
An algorithm for determining when we can apply our optimization follows:

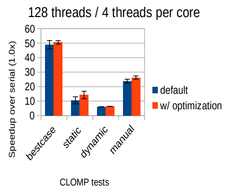
```
1. Mark all nodes (functions) as "may invoke task"
2. Convert the control-flow graph to a DAG of the
   strongly-connected components.
3. In reverse-topological order, take each SCC and
   mark all functions in the SCC as task-free iff:
    1. There are no task-generating API functions
       (determined with a list of blacklisted functions)
    2. There are no undefined functions (external
       calls)
    3. The descendant SCCs of this SCC are all task-
       free.
4. Repeat step 3 for all parent SCCs
```

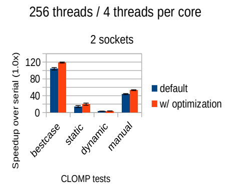
Results

Applying this optimization improves performance over an approximate optimization that marks every parallel region as task-free if there are no calls to task-generating functions anywhere in the program. This is because we can still optimize some parallel regions that don't contain tasks even if we see others that contain tasks. Below are results from the CLOMP benchmark:









The CLOMP tests perform a parallel computation with four different scheduling configurations and compare the speedup against a serial execution. The results show improvements of up to **94%** in execution speedup, depending on what scheduler is chosen.

Other Projects

I have also worked on a few bug fixes for the `clang` compiler for OpenMP support, as well as small improvements to the libcraymp runtime, and I am currently working on improved runtime debug messages for OpenMP offloading.