Using OpenMP for HEP Framework Algorithm Scheduling

Dr Christopher D Jones, Dr Patrick Gartung
CHEP 2019
4 November 2019

In partnership with:

This manuscript has been authored by Fermi Research Alliance, LLC under Contract No. DE-AC02-07CH11359 with the U.S. Department of Energy, Office of Science, Office of High Energy Physics.
Outline

Motivation

OpenMP Review

Demonstrator Frameworks

Experiment Setup

Results
Motivation

Why bother with OpenMP when already using Intel’s Threading Building Blocks?

HPC Centers
- Super Computing Centers traditionally use OpenMP for threading
- When communicating with HPC specialist, we are often asked about OpenMP
- Utilization of HPC centers for HEP will only increase over time

Need to either use OpenMP or have reason to not use
OpenMP Review

OpenMP is an extension to a compiler not a library
Uses compiler pragma statements
implementations of features vary considerably across compilers

OpenMP 4.5 Constructs

omp parallel
omp for
omp task
omp taskloop
OpenMP Construct: `omp parallel`

```
#pragma omp parallel
{ ... }
```

Starts threads used in the following block

Once assigned those threads can only be used by that parallel construct

At end of block the job waits till all assigned threads finish the block

number of threads for each parallel block is controlled by

env variable `OMP_NUM_THREADS` or calling `omp_set_num_threads`

Max number of threads for job is controlled by env variable `OMP_THREAD_LIMIT`
OpenMP Constructs: \texttt{omp for}

\begin{verbatim}
#pragma omp for
for(int i=0; i< N; ++i) { ... }
\end{verbatim}

Distributes iterations to threads associated with innermost parallel block

By default, calling thread waits till all iterations have completed
OpenMP Construct: nested parallel blocks

Support of concurrent nested parallel blocks is implementation defined
Also controlled by env variable OMP_NESTED or calling omp_set_nested

nested parallel blocks have as many threads as the outer blocks
Until max number of threads are reached
OpenMP Construct: nested parallel blocks — example 1

```
omp_set_num_threads(3);
#pragma omp parallel for
for(int i=0; i< 3; ++i){
  #pragma omp parallel for
  for(int j=0; j<3; ++j) {
    doWork(i,j);
  }
}
```

9 max threads per job

main thread waits till nested parallel finished
same as before except
6 max threads per job
finished threads cannot be used by other parallel blocks
OpenMP Construct: `omp task`

```
#pragma omp task
{
    ...
}
```

All code in the block is put into a task object

An **untied** task can be run by any thread of the innermost parallel section

When a task completes another task can be scheduled on the thread
The new task must be from the same parallel section
OpenMP Constructs: `omp taskloop`

```c
#pragma omp taskloop
for(int i=0; i< N; ++i){ ... }
```

Creates OpenMP tasks for the iterations

Calling thread may run other tasks while waiting for all `taskloop` tasks to end
I.e. implementations may do task stealing
Demonstrator Frameworks

Created simplified OpenMP, TBB and single threaded based frameworks

Frameworks can process multiple events concurrently

Work is done via Modules
  Modules generate data and put into events
  One Module can depend on data from other Modules
  Modules are wrapped in OpenMP or TBB tasks
  Module tasks only start once needed data are available

Modules may use parallel `for` constructs internally
  Allows testing of nested parallelism

Code available at `https://github.com/Dr15Jones/toy-mt-framework`
Experimental Setup

Compiled TBB and OpenMP frameworks with gcc 8 and clang 7
  Very different OpenMP 4.5 implementations

Created Module call graph that emulated CMS reconstruction
  Use same module dependencies
  Use module run times from 100 different events

Experiment varied
  Number of threads
    Number of concurrent events == number of threads
    Number of events processed in a job = Number of threads * 100
  Amount of module internal parallelism

Measurements done on an Intel KNL machine
Module Perfect Parallelism

All modules are concurrent capable

TBB results using gcc and clang are identical

Ran as many single-threaded jobs as number of threads

OpenMP and TBB have same results
One Serial Module with No Internal Parallelism

Simulate behavior of output
Serialize event access to the output module
All other modules are as before

Jobs quickly hit Ahmdal’s law limit
Serial Module with Internal Parallelism: Task Stealing

Allow output module to use parallelism
Use a for loop with 100 iterations

TBB uses `tbb::parallel_for`
does task stealing by default

OpenMP uses `taskloop`
clang does task stealing
gcc does not do task stealing

Task stealing hurts throughput
Serial Module with Internal Parallelism: No Task Stealing

Make all versions avoid task stealing

TBB use arenas

OpenMP uses `omp for`
Only way in API to guarantee no stealing
For each (max) number of threads
ran many jobs varying `omp_set_num_threads`
chose value with highest throughput

Even picking best working point for OpenMP, **TBB automatic behavior** gives best results
Conclusion

It is possible to create a HEP framework using OpenMP

Our investigation finds it would be less optimal than using TBB

Compiler implementation variations make portable performance hard

gcc taskloop does not do task stealing
clang taskloop does task stealing with no way to disable

OpenMP has compositibility difficulties

parallel blocks do not share threads
nested parallelism uses fixed allocation of threads
very hard to tune how many threads to use at each nested parallel level
Backup Slides
Task Stealing Problem

E.g. waiting thread steals a long running task

Can’t start `makeTasks` till stolen task finishes

```c
#pragma omp taskloop
for(int i=0; i< 2; ++i){
    doWork(i);
}
makeTasks();
```
Scanning Job Results for `omp for` usage

A selection of throughput vs `omp_set_num_threads` plots

Kept *maximum number of threads == number of concurrent events* for each measurement

![Throughput vs omp_set_num_threads plots for different maximum threads](image)