Lightweight Threaded Runtime Systems for OpenMP

Shintaro Iwasaki
Argonne National Laboratory, The University of Tokyo
Email: siwasaki@anl.gov, iwasaki@eidos.ic.i.u-tokyo.ac.jp
Outline of This Talk

- **BOLT**: a lightweight OpenMP library based on LLVM OpenMP.
  - It uses a lightweight user-level thread for OpenMP task and thread.
- **BOLT** won the Best Paper Award at PACT ’19[*]

**Features of BOLT:**

1. **Extremely lightweight OpenMP threads**
   that can efficiently handle nested parallelism.
2. **Tackle an interoperability issue of MPI + OpenMP task.**

This presentation will cover how to handle nested parallelism of BOLT.

- Please visit us!
  - [https://www.bolt-omp.org/](https://www.bolt-omp.org/)
  - or google “BOLT OpenMP”

[*] S. Iwasaki et al., “BOLT: Optimizing OpenMP Parallel Regions with User-Level Threads”, PACT ’19, 2019
Index

1. Introduction

2. User-level threads for OpenMP threads
   - Nested parallel regions and issues
   - Efficient adoption of ULTs
   - Evaluation

3. User-level threads for OpenMP tasks
   - OpenMP task and MPI operations
   - Tasking over ULT-aware MPI

4. Conclusions and future work
OpenMP: the Most Popular Multithreading Model

- **Multithreading** is essential for exploiting modern CPUs.

- **OpenMP** is a popular parallel programming model.
  - In the HPC field, OpenMP is most popular for multithreading.
    - 57% of DOE exascale applications use OpenMP [*].

- Not only user programs but also runtimes and libraries are parallelized by OpenMP.
  - Kokkos, RAJA, OpenBLAS, Intel MKL, SLATE, Intel MKL-DNN, FFTW3, ...

Unintentional Nested OpenMP Parallel Regions

- OpenMP parallelizes **multiple software stacks**.
- Nested parallel regions create OpenMP threads** exponentially**.

```c
// BLAS library
void dgemv(...) {
    #pragma omp parallel for
    for (i = 0; i < n; i++)
        dgemv_seq(data[n], i);
}

void dgemm(...):
    #pragma omp parallel for
    for (i = 0; i < n; i++)
        dgemm(matrix[n], ...);
```
Can We Just Disable Nested Parallelism?

- How to utilize nested parallel regions?
  - Enable nested parallelism: creation of exponential the number of threads
  - Disable nested parallelism: adversely decrease parallelism

- Example: strong scaling on massively parallel machines

Is the outer parallelism enough to feed work to all the cores???
Two Directions to Address Nested Parallelism

- Nested parallel regions have been known as a problem since OpenMP 1.0 (1997).
  - By default, OpenMP disables nested parallelism.[*].

- Two directions to address this issue:
  1. Use several work arounds implied in the OpenMP specification.
     => Not practical if users do not know parallelism at other software stacks.
  2. Instead of OS-level threads, use lightweight threads as OpenMP threads
     => It does not perform well if parallel regions are not nested (i.e., flat).
     • It does not perform well even when parallel regions are nested.

=> Need a solution to efficiently utilize nested parallelism.

[*] Since OpenMP 5.0, the default becomes “implementation defined”, while most OpenMP systems continue to disable nested parallelism by default.
BOLT: Lightweight OpenMP over ULT for Both Flat & Nested Parallel Regions

- We proposed BOLT, a ULT-based OpenMP runtime system, which performs best for both flat and nested parallel regions.

- Three key contributions:
  1. An in-depth performance analysis in the LLVM OpenMP runtime, finding several performance barriers.
  2. An implementation of thread-to-CPU binding interface that supports user-level threads.
  3. A novel thread coordination algorithm to transparently support both flat and nested parallel regions.
Index

1. Introduction

2. User-level threads for OpenMP threads
   - Nested parallel regions and issues
   - Efficient adoption of ULTs
   - Evaluation

3. User-level threads for OpenMP tasks
   - OpenMP task and MPI operations
   - Tasking over ULT-aware MPI

4. Conclusions and future work
Direction 1: Work around with OS-Level Threads (1/2)

### Several workarounds

1. **Disable** nested parallel regions
   
   \[ \text{OMP}_\text{NESTED}=\text{false}, \text{OMP}_\text{ACTIVE}\_\text{LEVELS}=\ldots \]
   
   - Parallelism is lost.

2. **Finely tune** numbers of threads
   
   \[ \text{OMP}_\text{NUM}\_\text{THREADS}={n}_1, {n}_2, {n}_3, \ldots \]
   
   - Parallelism is lost. Difficult to tune parameters.
Direction 1: Work around with OS-Level Threads (2/2)

- Workarounds (cont.)

3. Limit the total number of threads
   \(\text{OMP}_{-}\text{THREAD}\_\text{LIMIT}=n\text{ths}\)
   - Can adversely serialize parallel regions; doesn’t work well in practice.

4. Dynamically adjust # of threads
   \(\text{OMP}_{-}\text{DYNAMIC}=\text{true}\)
   - Can adversely serialize parallel regions; doesn’t work well in practice.

5. Use OpenMP task
   \(#\text{pragma omp task/taskloop}\)
   - Most codes use parallel regions. Semantically, threads != tasks.

- How about using lightweight threads for OpenMP threads?
Direction 2: Use Lightweight Threads

**User-Level Threads (ULTs)**

- User-level threads: threads implemented in user-space.
  - Manages threads without heavyweight kernel operations.

Thread scheduling (= context switching) involves **heavy system calls**.

User-level threads (ULTs) are running on Pthreads; scheduling is done by **user-level context switching** in user space.

Fork-Join Performance on KNL

---

[*] S. Seo et al. "Argobots: A Lightweight Low-Level Threading and Tasking Framework", TPDS '18, 2018
Using ULTs is Easy

- Replacing a Pthreads layer with a user-level threading library is a piece of cake.
  - Argobots\[*\] we used in this paper has the Pthreads-like API (mutex, TLS, ...), making this process easier.
  - The ULT-based OpenMP implementation is OpenMP 4.5-compliant (as far as we examined)

- Does the “baseline BOLT” perform well?

\[*\] S. Seo et al. "Argobots: A Lightweight Low-Level Threading and Tasking Framework", TPDS '18, 2018
Simple Replacement Performs Poorly

// Run on a 56-core Skylake server
#pragma omp parallel for num_threads(N)
for (int i = 0; i < N; i++)
    #pragma omp parallel for num_threads(28)
    for (int j = 0; j < 28; j++)
        comp_20000_cycles(i, j);

Nested Parallel Region (balanced)

- Faster than GNU OpenMP.
  - GCC
- So-so among ULT-based OpenMPs
  - MPC, OMPi, Mercurium
- Slower than Intel/LLVM OpenMPs.
  - Intel, LLVM

Popular Pthreads-based OpenMP

State-of-the-art ULT-based OpenMP

- GCC: GNU OpenMP with GCC 8.1
- Intel: Intel OpenMP with ICC 17.2.174
- LLVM: LLVM OpenMP with LLVM/Clang 7.0
- MPC: MPC 3.3.0
- OMPi: OMPi 1.2.3 and pthreads 1.0.4
- Mercurium: OmpSs (OpenMP 3.1 compat) 2.1.0 + Nanos++ 0.14.1
Index

1. Introduction

2. User-level threads for OpenMP threads
   - Nested parallel regions and issues
   - Efficient adoption of ULTs
   - Evaluation

3. User-level threads for OpenMP tasks
   - OpenMP task and MPI operations
   - Tasking over ULT-aware MPI

4. Conclusions and future work
Three Optimization Directions for Further Performance

The naïve replacement (BOLT (baseline)) does not perform well.

Need advanced optimizations

1. Solving scalability bottlenecks
2. ULT-friendly affinity
3. Efficient thread coordination
1. Solve Scalability Bottlenecks (1/2)

- **Resource management optimizations**
  1. **Divides a large critical section** protecting all threading resources.
     - This cost is negligible with Pthreads.
  2. **Enable multi-level caching of parallel regions**
     - Called “nested hot teams” in LLVM OpenMP.
1. Solve Scalability Bottlenecks (2/2)

- Thread creation optimizations


Serial Thread Creation (default LLVM OpenMP)

- Thread 1
- Thread 2
- Thread 3

Master (Thread 0)

Thread creation optimizations:

- Efficient resource management
- Scalable thread startup

// Run on a 56-core Skylake server
#pragma omp parallel for num_threads(L)
for (int i = 0; i < L; i++)
    #pragma omp parallel for num_threads(56)
    for (int j = 0; j < 56; j++)
        no_comp();

Nested Parallel Regions (no computation)

No computation to measure the pure overheads.

Binary Thread Creation

The critical path gets shorter.

Execution time [s]

- BOLT (baseline)
- Efficient resource management
- Scalable thread startup

Lower is better
2. Affinity: How to Implement Affinity for ULTs

- OpenMP 4.0 introduced *place* and *prod_bind* for affinity.
  - OS-level thread-based libraries (e.g., GNU OpenMP) use CPU masks.
- BOLT (baseline) ignored affinity (still standard compliant).
- However, affinity should be useful to
  1. improve locality and 2. reduce queue contentions.
  - Note: ULT runtimes use shared queues + random work stealing.
- *How to implement place over ULTs?*

```c
// OMP_PLACES={0,1},{2,3},{4,5},{6,7}
// OMP_PROC_BIND=spread
#pragma omp parallel for num_threads(4)
for (i = 0; i < 4; i++)
    comp(i);
```
Implementation: Place Queue

- **Place queues** can implement OpenMP affinity in BOLT.

```c
// OMP_PLACES={0,1},{2,3},{4,5},{6,7}
// OMP_PROC_BIND=spread
#pragma omp parallel for num_threads(4)
for (i = 0; i < 4; i++)
    comp(i);
```

- **Problem:** OpenMP affinity setting is too deterministic.
OpenMP Affinity is Too Deterministic

- Affinity (or bind-var) is once set, all the OpenMP threads created in the descendant parallel regions are bound to places.

```
// OMP_PLACES={0,1},{2,3},{4,5},{6,7}
// OMP_PROC_BIND=spread
#pragma omp parallel for num_threads(8)
for (int i = 0; i < 8; i++)
    #pragma omp parallel for num_threads(8)
    for (int j = 0; j < 8; j++)
        comp(i, j);
```

The OpenMP specification writes so.

- Promising direction: scheduling innermost threads with unbound random work stealing.
Proposed New PROC_BIND: “unset”

**OMP_WAIT_POLICY=unset**: reset the affinity setting of the specified parallel region.

(Detailed: The `unset` thread affinity policy resets the `bind-var ICV` and the `place-partition-var ICV` to their implementation defined values and instructs the execution environment to follow these values.)

```
// OMP_PLACES={0,1},{2,3},{4,5},{6,7}  
// OMP_PROC_BIND=spread
#pragma omp parallel for num_threads(8)
for (int i = 0; i < 8; i++)
    #pragma omp parallel for num_threads(8)
    for (int j = 0; j < 8; j++)
        comp(i, j);
```

- This **scheduling flexibility** gives higher performance.

Random work stealing for innermost threads.

### Graph

- **Execution time [s]**
- **# of outer threads (N)**
- Lower is better
3. Flat Parallelism: Poor Performance

- BOLT should perform as good as the original LLVM OpenMP.

- Optimal OMP_WAIT_POLICY for GCC/Intel/LLVM improves performance of flat parallelism.

```
#pragma omp parallel for num_threads(56)
for (int i = 0; i < 56; i++)
    #pragma omp parallel for num_threads(56)
    for (int j = 0; j < 56; j++) no_comp(i, j);
```

```
#pragma omp parallel for num_threads(56)
for (int i = 0; i < 56; i++)
    no_comp(i);
```

![Execution time comparison](image)
Active Waiting Policy for Flat Parallelism

- Active waiting policy improves performance of flat parallelism by busy-wait based synchronization.

- If active, Pthreads-based OpenMP busy-waits for the next parallel region.

- BOLT on the other hand yields to a scheduler on fork-and-join (~ passive).

```c
for (int iter = 0; iter < n; iter++) {
    #pragma omp parallel for num_threads(4)
    for (int i = 0; i < 4; i++)
        comp(i);
}
```

* If passive, after completion of work, threads sleep on a condition variable.

Busy wait is faster than lightweight user-level context switch!

OMP_WAIT_POLICY =<active/passive>
Implementation of Active Policy in BOLT

- If active, busy-waits for next parallel regions.
- If passive, relies on ULT context switching.

ULT threads are not preemptive, so BOLT periodically yields to a scheduler in order to avoid the deadlock (especially when # of OpenMP threads > # of schedulers).
Performance of Flat and Nested

<table>
<thead>
<tr>
<th>Execution time [us]</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOLT</td>
</tr>
<tr>
<td>1E+0</td>
</tr>
</tbody>
</table>

**Nested (passive)**

- MPC serializes nested parallel regions, so it’s fastest.

**Flat (active)**

- As BOLT didn’t, MPC ... OMPI do not implement the active policy.

```
#pragma omp parallel for num_threads(56)
for (int i = 0; i < 56; i++)
    #pragma omp parallel for num_threads(56)
    for (int j = 0; j < 56; j++)
        no_comp(i, j);
```

```
#pragma omp parallel for num_threads(56)
for (int i = 0; i < 56; i++)
    no_comp(i);
```
Penalty of the Opposite Wait Policy

- How to coordinate threads significantly affects the overheads.
  - Large performance penalty discourages users from enabling nesting.

- Is there a good algorithm to transparently support both flat and nested parallelism?

```c
#pragma omp parallel for num_threads(56)
for (int i = 0; i < 56; i++)
  #pragma omp parallel for num_threads(56)
  for (int j = 0; j < 56; j++) no_comp(i, j);
```

```c
#pragma omp parallel for num_threads(56)
for (int i = 0; i < 56; i++)
  #pragma omp parallel for num_threads(56)
  for (int j = 0; j < 56; j++) no_comp(i);
```
Busy Waiting in Both Active/Passive Algorithms

- Though in both active and passive cases, they enter busy-waits after the completion of threads.
  - Can we merge it to perform both scheduling and flag checking?
Algorithm: Hybrid Wait Policy

- **Hybrid**: execute flag check and queue check alternately.
  - [flat]: a thread does not go back to a scheduler.
  - [nested]: another available ULT is promptly scheduled.

```c
void omp_thread() {
    RESTART_THREAD:
    comp();
    while (time_elapsed() < KMP_BLOCKTIME) {
        if (team->next_parallel_region_flag) {
            goto RESTART_THREAD;
            ULT_t *ult = get_ULT_from_queue (parent_scheduler);
        if (ult != NULL)
            return_to_sched_and_run(ult);
    }
}
```

This technique is not applicable to OS-level threads since the scheduler is not revealed.
- BOLT (hybrid wait policy) is always most efficient in both flat and nested cases.
  - We suggest a new keyword “auto” so that the runtime can choose the implementation.
Summary of the Design

- Just using ULT is insufficient.
  => Three kinds of optimizations:
  1. Address **scalability bottlenecks**
  2. ULT-friendly **affinity**
  3. Hybrid wait policy for flat and nested parallelisms

```c
// Run on a 56-core Skylake server
#pragma omp parallel for num_threads(L)
for (int i = 0; i < L; i++)
  #pragma omp parallel for num_threads(56)
  for (int j = 0; j < 56; j++)
      no_comp();
```
Index

1. Introduction

2. User-level threads for OpenMP threads
   – Nested parallel regions and issues
   – Efficient adoption of ULTs
   – Evaluation

3. User-level threads for OpenMP tasks
   – OpenMP task and MPI operations
   – Tasking over ULT-aware MPI

4. Conclusions and future work
Microbenchmarks

```c
// Run on a 56-core Skylake server
#pragma omp parallel for num_threads(L)
for (int i = 0; i < L; i++) {
    #pragma omp parallel for num_threads(28)
    for (int j = 0; j < 28; j++)
        comp_20000_cycles(i, j);
}

// Run on a 56-core Skylake server
#pragma omp parallel for num_threads(56)
for (int i = 0; i < 56; i++) {
    int work_cycles = get_work(i, alpha);
    #pragma omp parallel for num_threads(56)
    for (int j = 0; j < 56; j++)
        comp_cycles(i, j, work_cycles);
}
```

**alpha** makes the computation size random, while keeping the total problem size.

(Ideal): theoretical lower bound under perfect scalability.
Microbenchmarks: vs. taskloop

// Run on a 56-core Skylake server
#pragma omp parallel for num_threads(56)
for (int i = 0; i < L; i++) {
    #pragma omp taskloop grainsize(1)
    for (int j = 0; j < 28; j++)
        comp_20000_cycles(i, j);
}

Parallel regions of BOLT are as fast as taskloop!
Evaluation: KIFMM

- **KIFMM[^]**: highly optimized N-body solver
  - N-body solver is one of the heaviest kernels in astronomy simulations.
- Multiple layers are parallelized by OpenMP.
  - BLAS and FFT.
- We focus on the upward phase in KIFMM.

```c
for (int i = 0; i < max_levels; i++)
    #pragma omp parallel for
    for (int j = 0; j < nodecounts[i]; j++) {
        [...];
        dgemv(...); // dgemv() creates a parallel region.
    }
```

**Performance: KIFMM**

void kifmm_upward():
   for (int i = 0; i < max_levels; i++)
      #pragma omp parallel for num_threads(56)
      for (int j = 0; j < nodecounts[i]; j++) {
         [...];
         dgemv(...); // creates a parallel region.
      }

void dgemv(...): // in MKL
   #pragma omp parallel for num_threads(N)
   for (int i = 0; i < [...]; i++)
      dgemv_sequential(...);

- **Experiments on Skylake 56 cores.**
  - # of threads for the outer parallel region = 56
  - # of threads for the inner parallel region = N (changed)

- **Two important results:**
  - N=1 (flat): performance is almost the same.
  - N>1 (nested): BOLT further boosts performance.
Summary: ULT-based OpenMP Threads

- Nested OpenMP parallel regions are commonly seen in complicated software stacks.
  => Demand for efficient OpenMP runtimes to exploit both flat and nested parallelism.

- BOLT: an lightweight OpenMP library over ULT.
  - Simply using ULTs is insufficient:
    - Solve scalability bottlenecks in the LLVM OpenMP runtime
    - ULT-friendly affinity implementation
    - Hybrid thread coordination technique to transparently support both flat and nested parallel regions.

- BOLT achieves unprecedented performance for nested parallel regions without hurting the performance of flat parallelism.
Index

1. BOLT is OpenMP over lightweight threads
   - What is a user-level thread (ULT)?
2. User-level threads for OpenMP threads
   - Nested parallel regions and issues
   - Efficient adoption of ULTs
   - Evaluation
3. User-level threads for OpenMP tasks
   - OpenMP task and MPI operations
   - Tasking over ULT-aware MPI
4. Conclusions and future work
Argobots Ecosystem

- Argobots-Aware MPICH
- Charm++ Applications
- Communication libraries
- Argobots runtime

- Fused ULT 1
- Fused ULT 2
- Fused ULT N
- Argobots ES
- Cilk Bots

- OpenMP (BOLT)
- XcalableMP
- Intel DAOS

- Origin
- Mercury RPC

- PaRSEC

- Open MPI
- Progress Engine

- #pragma xmp loop for (…)

- Global Thread Team

- Task-Pool

- Argobots ES

- OmpSs

- Cilk "Worker"

- Open MPI over Argobots
BOLT in ECP SOLLVE

ECP OpenMP Working Groups
- Application Expert
- Compiler & RT Expert.
- OpenMP Language Rep
- Vendor representative
- Tester & Integrator

OpenMP Extensions
- Memory Hierarchies
- Deep Copy
- Performance Portability
- Tasking
- Heterogeneous Devices
- MPI Interoperability

OpenMP ARB Standards

Vendors

ECP Tools Dev.

ECP Applications (Apps, Libraries)
- Clang C/C++ FE
- Flang (PGI Fortran FE)

LLVM Compiler

Runtime Layer (OpenMP RTL, BOLT, Argobots)

ECP Operating Systems

NUMA Mem. Interconnect
- CPUs
- HBM
- DRAM
- NVM

Exascale Nodes on Multiple Architectures

ECP Proposed Contributions
- Application Feedback
- Programming Model Extensions
- Compiler & Runtime Implementation
- HW/SW Co-design

Interactions
- Continuous Testing and Integration
  - Unit Testing
  - Validation Suite
  - Multi-platform Testing

Compiler Extensions

OpenMP Runtime Extensions
How to Build BOLT?

1. Use the latest version ([https://github.com/pmodels/bolt](https://github.com/pmodels/bolt))
   - Please follow the instruction.

   ```bash
   git clone https://github.com/pmodels/bolt.git $BOLT_DIR
   cd $BOLT_DIR
   ## to use the very latest version:
   # git checkout latest
   git submodule update --init
   mkdir build && cd build
   cmake ../
   cmake ./ -DCMAKE_INSTALL_PREFIX=BOLT_INSTALL_DIR \
       -DCMAKE_BUILD_TYPE=Release -DLIBOMP_USE_ARGOBOTS=on
   make -j install
   ```

2. Use Spack ([https://github.com/spack/spack](https://github.com/spack/spack))

   ```bash
   spack install bolt
   ```
BOLT: High Compatibility with Existing Libraries

- BOLT keeps compatibility with the LLVM OpenMP interface.
  - Several compilers (GCC, Intel Compilers, and Clang/LLVM) can be used as a frontend.
    - i.e., BOLT does not require a special compiler.

- No need to recompile existing applications and libraries for BOLT!

Note: some existing proposals require special compilers.
How to Use BOLT?

- No recompilation needed. Just change the runtime library.

- [Recommended] Set LD_LIBRARY_PATH

  ```
  $ LD_LIBRARY_PATH="$BOLT/install/lib:${LD_LIBRARY_PATH}" ./prog
  ```

  Please check if BOLT is loaded by `ldd`:

  ```
  $ LD_LIBRARY_PATH="$BOLT/install/lib:${LD_LIBRARY_PATH}" ldd ./prog
  ```

  ```
  linux-vdso.so.1 => (0x00007fff3bbbe000)
  libm.so.6 => /lib64/libm.so.6 (0x00007f6e9fc29000)
  libiomp5.so => /home/user/bolt/install/lib/libiomp5.so (0x00007f6e9f994000)
  ```

- [Fallback] Set LD_PRELOAD

  ```
  # GCC
  $ LD_PRELOAD="$BOLT/install/lib/libgomp.so:${LD_PRELOAD}" ./prog
  ```

  ```
  # Intel C/C++ Compilers
  $ LD_PRELOAD="$BOLT/install/lib/libiomp5.so:${LD_PRELOAD}" ./prog
  ```

  ```
  # Clang/LLVM
  $ LD_PRELOAD="$BOLT/install/lib/libomp.so:${LD_PRELOAD}" ./prog
  ```

  If you cannot find it, LD_LIBRARY_PATH does not work. It often happens if you use GCC as a frontend.
BOLT: Supported Features

- BOLT supports OpenMP 4.5 with a few exceptions.
  - As the original LLVM OpenMP supports.

- BOLT support includes ...
  - Basic parallel regions (parallel for/section)
  - Tasks: task, taskloop, task depend
  - Target offloading: (e.g., offload computation to a GPU device)
    - BOLT does not affect the GPU performance
  - Synchronization: single/master/barrier/order, omp_lock
  - SIMD directives: supported by compilers

- BOLT currently does not support ...
  - OMPT & OMPD (though they are OpenMP 5.0 features)
  - Task cancellation
Thank You for Listening!

- **BOLT[**]: an lightweight OpenMP library over Argobots.
  - BOLT achieves unprecedented performance for nested parallel regions without hurting the performance of flat parallelism.

- It is available at GitHub / Spack
  1. BOLT repository ([https://github.com/pmodels/bolt](https://github.com/pmodels/bolt))

```sh
spack install bolt
```

[**] S. Iwasaki et al., “BOLT: Optimizing OpenMP Parallel Regions with User-Level Threads”, PACT ’19, 2019