BOLT: Optimizing OpenMP Parallel Regions with User-Level Threads

Shintaro Iwasaki†, Abdelhalim Amer‡, Kenjiro Taura†, Sangmin Seo‡, Pavan Balaji‡

†The University of Tokyo
‡Argonne National Laboratory

Email: iwasaki@eidos.ic.i.u-Tokyo.ac.jp, siwasaki@anl.gov
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
#include <omp.h>

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_seq(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

```c
#pragma omp parallel for
for (i = 0; i < n; i++)
    comp(cells[i], ...);

void comp(...):
    [...];
#pragma omp parallel for
for (i = 0; i < n; i++)
    [...];
```

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. Existing Approaches
   - OS-level thread-based approach
   - User-level thread-based approach
     • What is a user-level thread (ULT)?

3. BOLT for both Nested and Flat Parallelism
   - Scalability optimizations
   - ULT-aware affinity (proc_bind)
   - Thread coordination (wait_policy)

4. Evaluation

5. Conclusion
### Direction 1: Work around with OS-Level Threads (1/2)

#### Several workarounds

1. **Disable** nested parallel regions
   - `OMP_NESTED=false, OMP_ACTIVE_LEVELS=...`
   - Parallelism is lost.

2. **Finely tune** numbers of threads
   - `OMP_NUM_THREADS=nth1,nth2,nth3,...`
   - 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**
     \( (\text{OMP\_THREAD\_LIMIT=\text{nths}}) \)
     - Can adversely serialize parallel regions; doesn’t work well in practice.
  4. **Dynamically adjust # of threads**
     \( (\text{OMP\_DYNAMIC=true}) \)
     - 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.

Naïve Pthreads

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

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

Kernel (OS)

Fork-Join Performance on KNL

> 350x

[Naïve Pthreads]

[*] S. Seo et al. "Argobots: A Lightweight Low-Level Threading and Tasking Framework", TPDS '18, 2018
Solution 2: Use User-Level Threads

- The idea of ULTs is not new (back to <90s).

- Several ULT-based OpenMP systems have been proposed.
  - NanosCompiler [1], Omni/ST [2], OMPI [3], MPC [4], ForestGOMP [5], OmpSs (OpenMP compatible mode) [6], LibKOMP [7] ...


- However, these runtimes do not perform well for several reasons.
  - Lack of OpenMP specification-aware optimizations
  - Lack of general optimizations

For apples-to-apples comparison, we will focus on the ULT-based LLVM OpenMP.
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 psthreads 1.0.4
Mercurium: OmpSs (OpenMP 3.1 compat) 2.1.0 + Nanos++ 0.14.1
Index

1. Introduction
2. Existing Approaches
   – OS-level thread-based approach
   – User-level thread-based approach
     • What is a user-level thread (ULT)?
3. BOLT for both Nested and Flat Parallelism
   – Scalability optimizations
   – ULT-aware affinity (proc_bind)
   – Thread coordination (wait_policy)
4. Evaluation
5. Conclusion
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

// 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)
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)

```
// 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();
```

Binary Thread Creation

- The critical path gets shorter.

Execution time [s] vs # of outer threads (L)

- 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);
```

- 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.
(TM: 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.)

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

```c
// 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);
```

**Random work stealing for innermost threads.**

![Diagram showing scheduling and execution time comparison](image)

**Execution time [s]**

- **BOLT (baseline)**
- + Efficient resource management
- ++ Scalable thread startup
- +++ Bind=spread
- ++++ Bind=spread,unset

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.
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).

```
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!
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

```
#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);
```

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

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

Lower is better
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?
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.
Performance of Hybrid: Flat and Nested

- 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.

#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);
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

- Our work solely focuses on OpenMP, while some of our techniques are generic:
  - Place queues for affinity of ULTs
  - Hybrid thread coordination for runtimes that have parallel loop abstraction.

```
// 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();
```
1. Introduction
2. Existing Approaches
   - OS-level thread-based approach
   - User-level thread-based approach
     • What is a user-level thread (ULT)?
3. BOLT for both Nested and Flat Parallelism
   - Scalability optimizations
   - ULT-aware affinity (proc_bind)
   - Thread coordination (wait_policy)
4. Evaluation
5. Conclusion
Microbenchmarks

```
// 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);
}
```

```
// 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);
}
```

- Parallel regions of BOLT are **as fast as taskloop!**
Evaluation: Use Case of Nested Parallel Regions

- The number of threads for outer loops is **usually set to # of cores**.
  - i.e., if not nested, oversubscription does not happen.
- However, many layers are **OpenMP parallelized**, which can unintentionally result in nesting.
- We will show two examples.
Evaluation 1: 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.

\begin{verbatim}
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.
  }
\end{verbatim}

\footnote{A. Chandramowlishwaran et al., "Brief Announcement: Towards a Communication Optimal Fast Multipole Method and Its Implications at Exascale", SPAA '12, 2012}
### Performance: KIFMM

**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.

```c
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(...);
```

![Graph showing relative performance vs. number of inner threads (N)](chart.png)

**Notes:**
- Different Intel OpenMP configurations: nobind(=false), true, close, spread: proc_bind dyn: MKL_DYNAMIC=true
- Note that other parameters are hand tuned (see the paper).
Evaluation 2: FFT in QBox

- **Qbox**: first-principles molecular dynamics code.

- **We focus on the FFT computation part.**

- We extracted this FFT kernel and change the parameters based on the gold benchmark.

```c
// FFT backward
#pragma omp parallel for
for (int i = 0; i < num / nprocs; i++)
    fftw_execute(plan_2d, ...);

void fftw_execute(...): // in FFTW3
    [...];
#pragma omp parallel for num_threads(N)
for (int i = 0; i < [...]; i++)
    fftw_sequential(...);
```

### Performance: FFTW3

`// FFT backward
#pragma omp parallel for
for (int i = 0; i < num / nprocs; i++)
  fftw_execute(plan_2d, ...);

void fftw_execute(...): // in FFTW3
  [...];
#pragma omp parallel for num_threads(N)
for (int i = 0; i < [...]; i++)
  fftw_sequential(...);`

- nprocs = # of MPI nodes
- num (and fftw size) is proportional to # of atoms.

Experiments on KNL 7230 64 cores.
- # of threads for the outer parallel region = 64
- # of threads for the inner parallel region = N (changed)

- **N=1 (flat):** performance is almost the same.
- **N>1 (nested):** BOLT further increased performance.

**Intel OpenMP configurations:** nobind(=false), true, close, spread: proc_bind, dyn: OMP_DYNAMIC=true

X axis: # of inner threads (N)
Y axis: relative performance (BOLT + N=1: 1.0)

More beneficial for nested parallel regions. => Strong scaling
Index

1. Introduction

2. Existing Approaches
   – OS-level thread-based approach
   – User-level thread-based approach
     • What is a user-level thread (ULT)?

3. BOLT for both Nested and Flat Parallelism
   – Scalability optimizations
   – ULT-aware affinity (proc_bind)
   – Thread coordination (wait_policy)

4. Evaluation

5. Conclusion
Summary of this Talk

- 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.
Thank you for listening!

- **BOLT**: [http://www.bolt-omp.org](http://www.bolt-omp.org)

- **Q&A (as a software)**:
  - What is the goal of the BOLT project?
    - Improve OpenMP by ULTs:
      - 1. enrich OpenMP tasking features with least overheads,
      - 2. minimizing overheads of OpenMP threads, and 3. more.
  - How to use it?
    - BOLT is a runtime library: no special compiler is required.
      
      GCC/ICC/Clang + LD_LIBRARY_PATH+=${BOLT_INSTALL_PATH} works.
  - Is BOLT stable?
    - Regularly checked with LLVM OpenMP tests (GCC 8.x, ICC 19.x, and Clang 10.x)
  - What OpenMP features are supported?
    - OpenMP 4.5 including task, task depend, and offloading.

Acknowledgment

This research was supported by the Exascale Computing Project (17-SC-20-SC), a joint project of the U.S. Department of Energy's Office of Science and National Nuclear Security Administration, responsible for delivering a capable exascale ecosystem, including software, applications, and hardware technology, to support the nation's exascale computing imperative. This research is in particular its subproject on Scaling OpenMP with LLVM for Exascale performance and portability (SOLLVE).

BOLT is part of the ECP SOLLVE project: [https://www.bnl.gov/compsci/projects/SOLLVE/](https://www.bnl.gov/compsci/projects/SOLLVE/)

Artifact: [https://zenodo.org/record/3372716](https://zenodo.org/record/3372716) (DOI: 10.5281/zenodo.3372716)

Future work:
- Enhance task scheduling
- MPI+OpenMP interoperability

Much engineering efforts for ABI compatibility and stability.