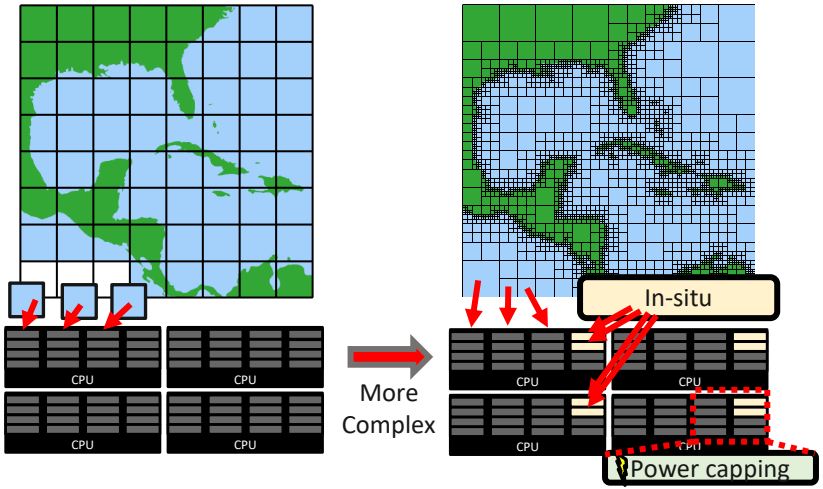
OpenMP®

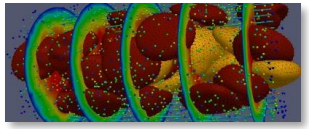
SC'20 Booth Talk Series

BOLT: A Lightweight and Highly Interoperable OpenMP Runtime System

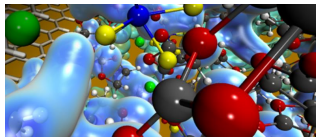
Shintaro Iwasaki (Argonne National Laboratory)
siwasaki@anl.gov

Exploiting Parallelism for Efficient Computing

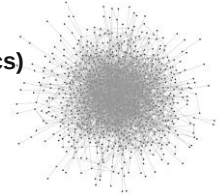




Biology
(heart murmur simulation)



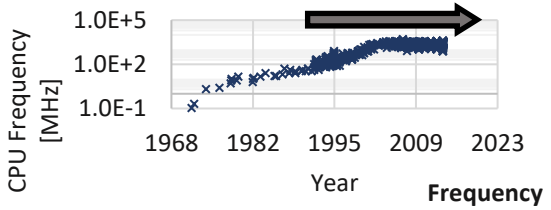
Chemistry
(molecular dynamics)

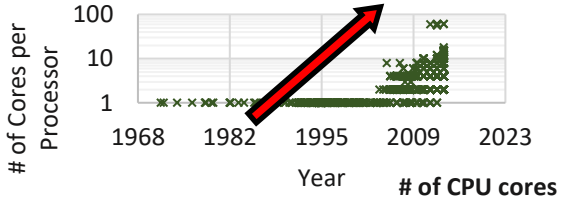


Graph Analytics

[Right] Intel Xeon Phi (Knights Landing) 72 cores, 288 HWTs
(https://software.intel.com/en-us/articles/what-disclosures-has-intel-made-about-knights-landing)

[Left] ARM ThunderX2 up to 32 cores, 128 HWTs
(https://www.servethehome.com/cavium-thunderx2-review-benchmarks-real-arm-server-option/)



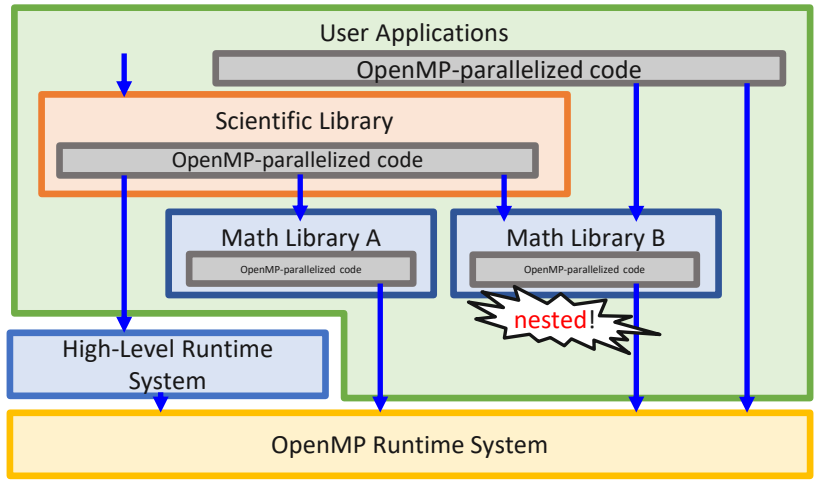


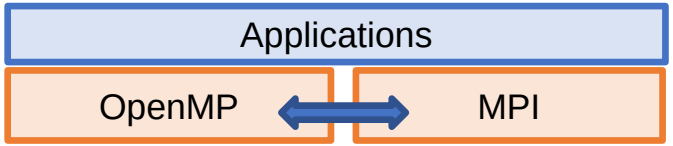
CPU DB: (http://cpudb.stanford.edu/)

- Highly parallel compute resources
- Complex workloads across several software stacks
- How about OpenMP?

Efficient Thread/Task Management for OpenMP + X?

- OpenMP + OpenMP
  - Nested parallel regions
  - Creation of OpenMP threads at each level of parallel regions can exponentially increase the total number of threads.





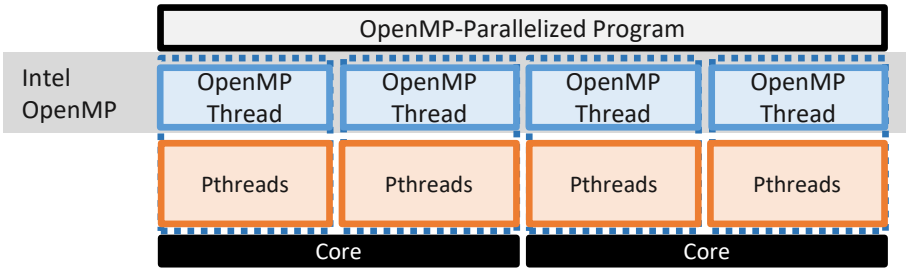
- OpenMP + MPI
  - Poor performance of multithreaded MPI because of heavy lock contention.
  - MPI + OpenMP tasks?

- OpenMP + other parallel runtime systems …

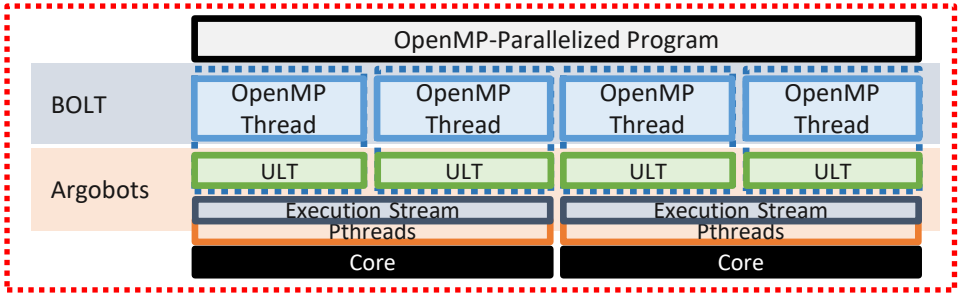
BOLT: OpenMP over Argobots [*]

[*] S. Iwasaki et al. "BOLT: Optimizing OpenMP Parallel Regions with User-Level Threads", PACT '19, 2019 (BEST PAPER)

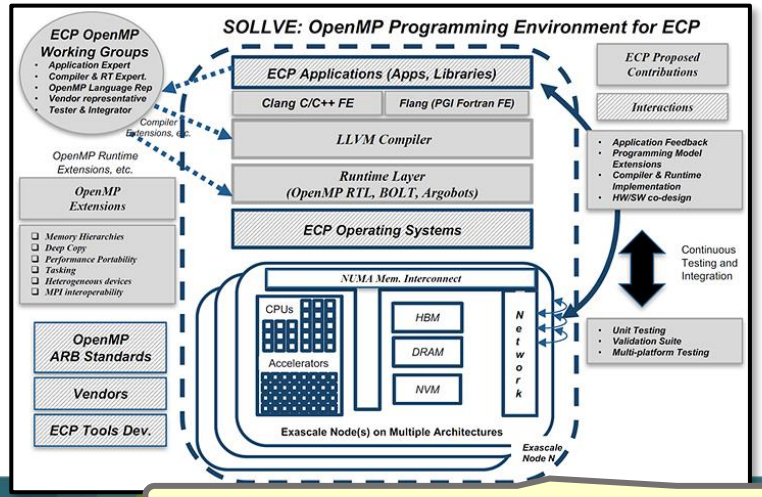
- Use lightweight Argobots threads for OpenMP threads and tasks



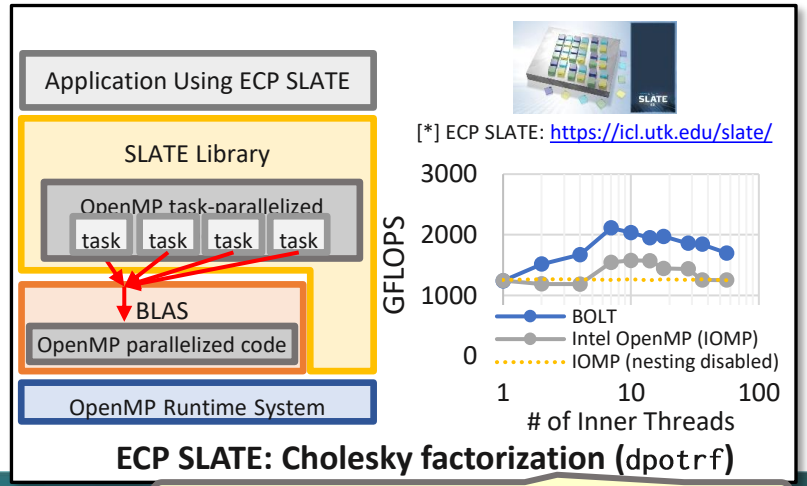
Traditional OpenMP (Intel OpenMP)



BOLT: LLVM OpenMP over Argobots



BOLT is part of the ECP SOLLVE project.



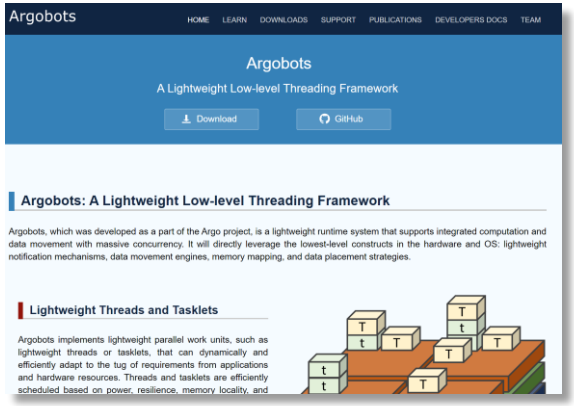
ECP SLATE: Cholesky factorization (`dpotrf`)

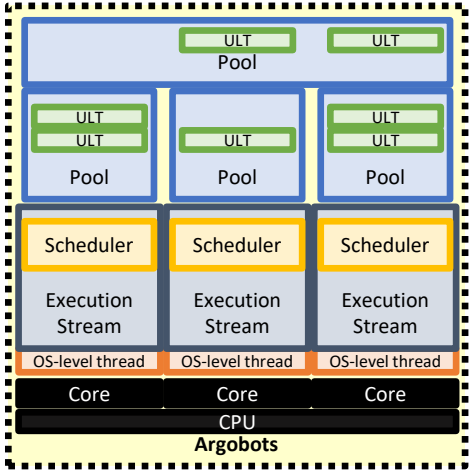
BOLT can exploit nested parallel regions.

Design of BOLT

Argobots: A Low-Level Lightweight Threading Library

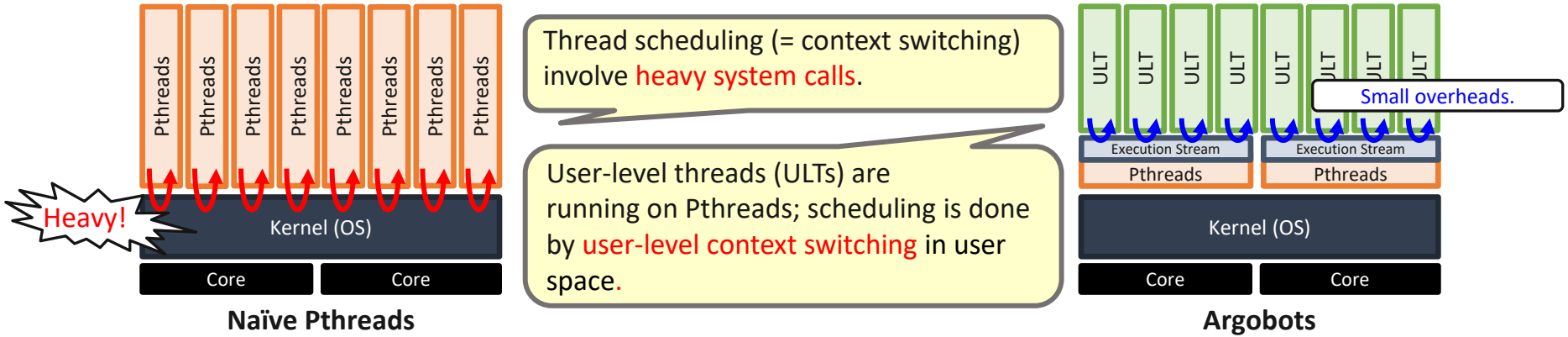
- Argobots is an open-source project
  - URL: https://argobots.org/
  - Maintained by Argonne National Laboratory
    - Collaborators: UIUC, Univ. of Tennessee, PNNL, Intel, UTokyo, Riken, …
    - Supported by DOE
- Unleashes user-level threads (ULTs)
- Provides for the future scalable systems.
  - Extremely lightweight "thread" implementation
  - Rich and powerful threading capabilities
  - Low-level customizability



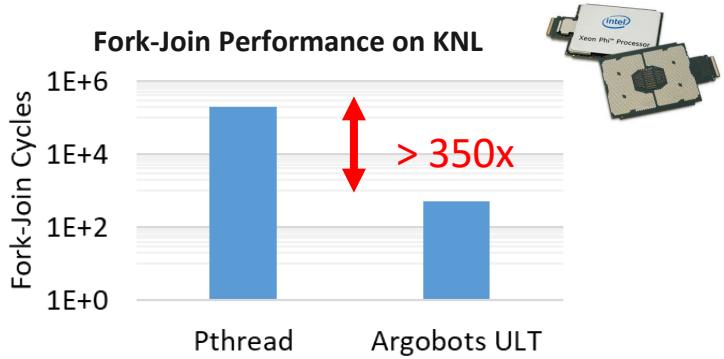


What is a User-Level Thread (ULT)?

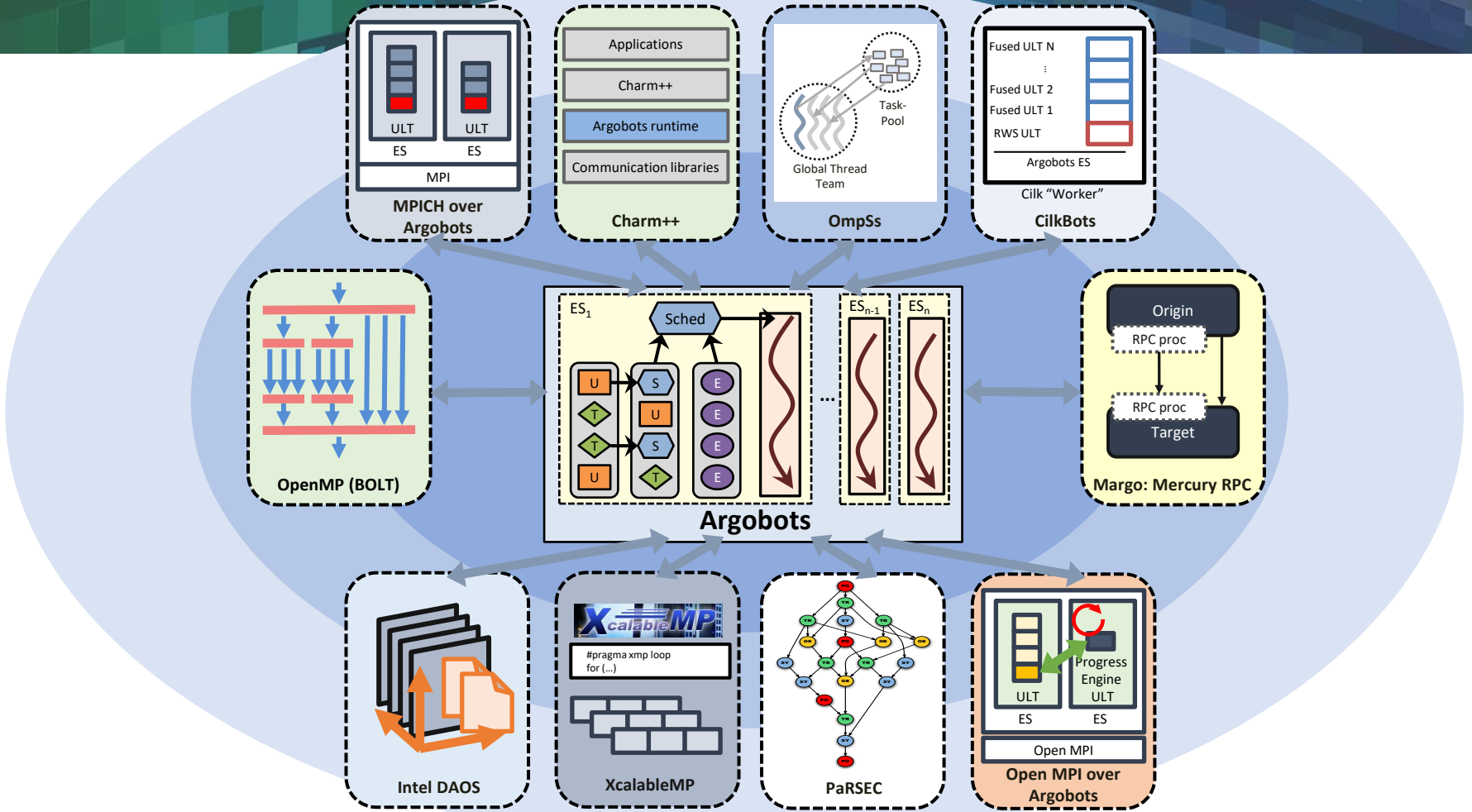
- A user-level thread (ULT) implements all threading operations in user space.



- Advantages of ULTs:
  1. Lightweight thread with low context-switching overhead
  2. Multiple ULTs can be mapped to a single OS-level thread
  3. Users can control scheduling



Ecosystem of Argobots

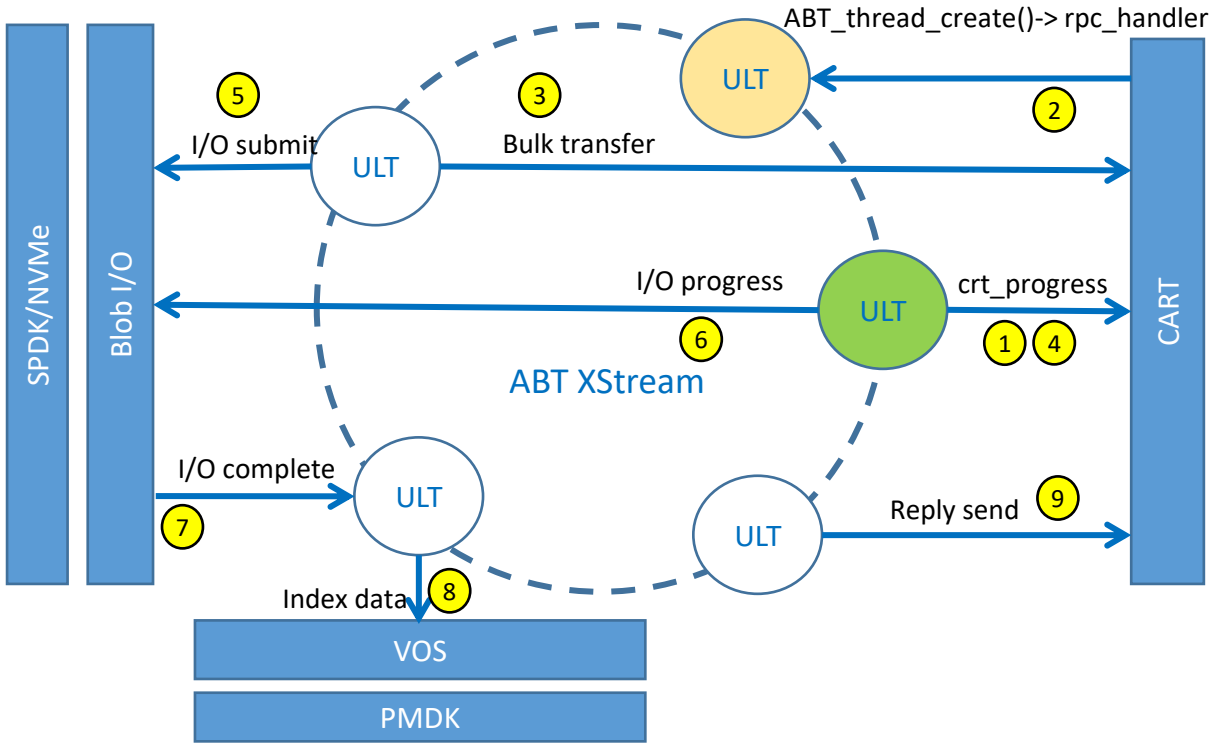


Use Case of Argobots (1) Intel DAOS I/O Stack

SOLUTION BRIEF
Distributed Asynchronous Object Storage (DAOS)
Intel® Optane™ Technology
DAOS: Revolutionizing High-Performance Storage with Intel® Optane™ Technology

- Intel **D**istributed **A**pplication **O**bject **S**torage
- Exascale I/O stack
- Argobots is used in server's I/O handling
  - Each I/O is handled by one ULT
  - Waiting for I/O => yield CPU and switch to another ULT

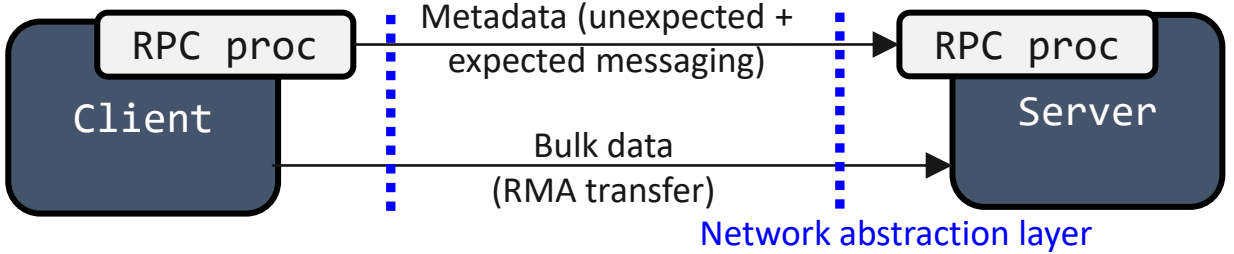
https://github.com/daos-stack/daos



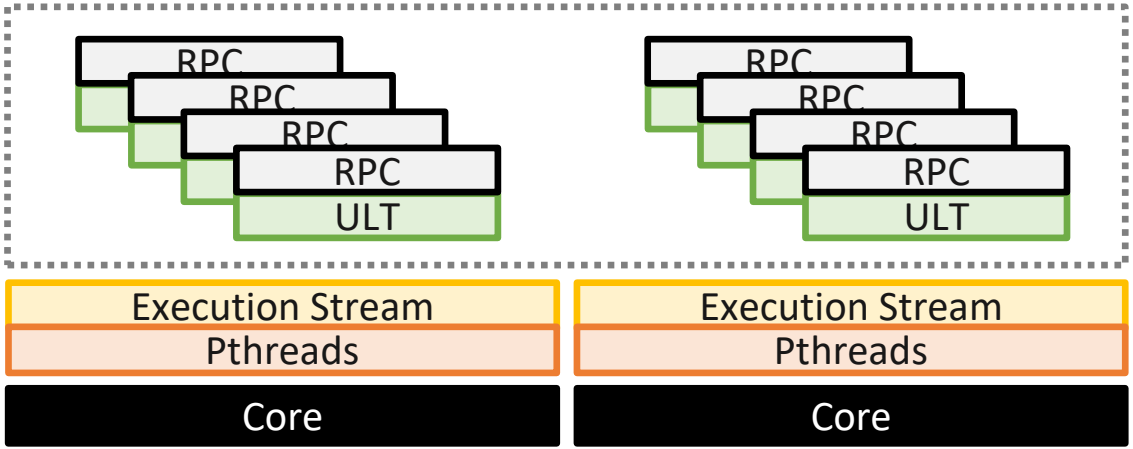
Use Case of Argobots (2) Margo: Mercury over Argobots

- Margo is a lightweight communication library supporting RPC and RDMA.

https://mochi.readthedocs.io/en/latest/margo.html

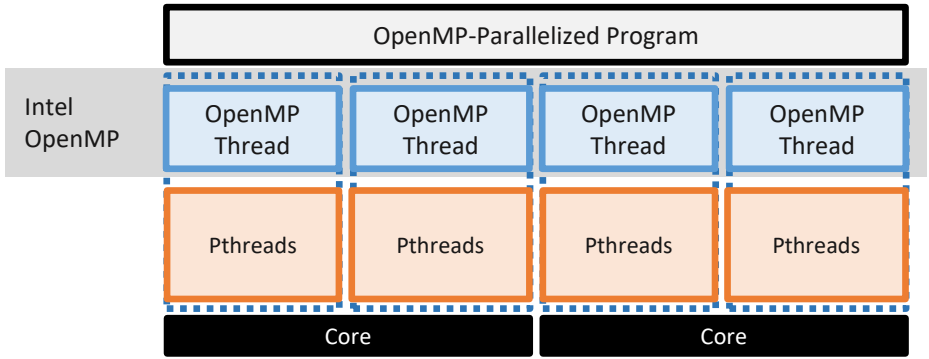


- Margo utilizes Argobots to progress communications instead of user's ugly progress loop and callbacks.

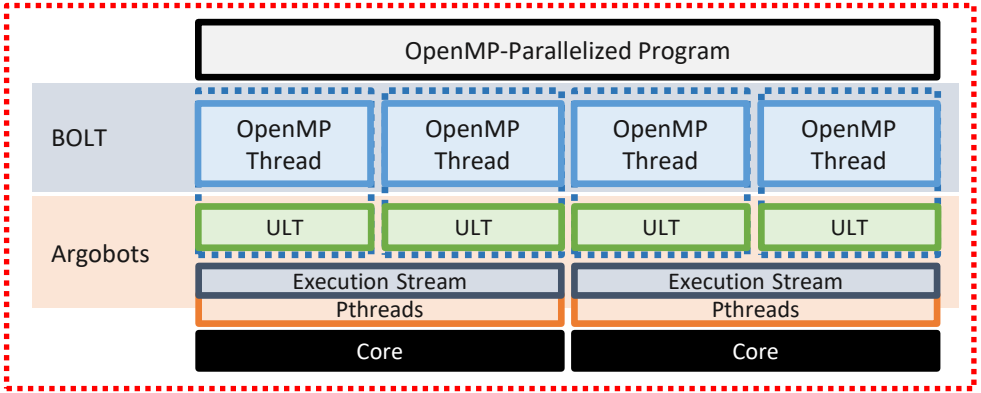


BOLT: BOLT is OpenMP over Lightweight Threads

- BOLT: an OpenMP library over Argobots
  - Based on LLVM OpenMP 10.0
  - Provides most of the latest OpenMP features.
  - Maps both OpenMP threads and tasks to Argobots ULTs.
    - Tiny threading overheads.
  - High ABI compatibility with LLVM/Intel/GNU OpenMP



Traditional OpenMP (Intel OpenMP)

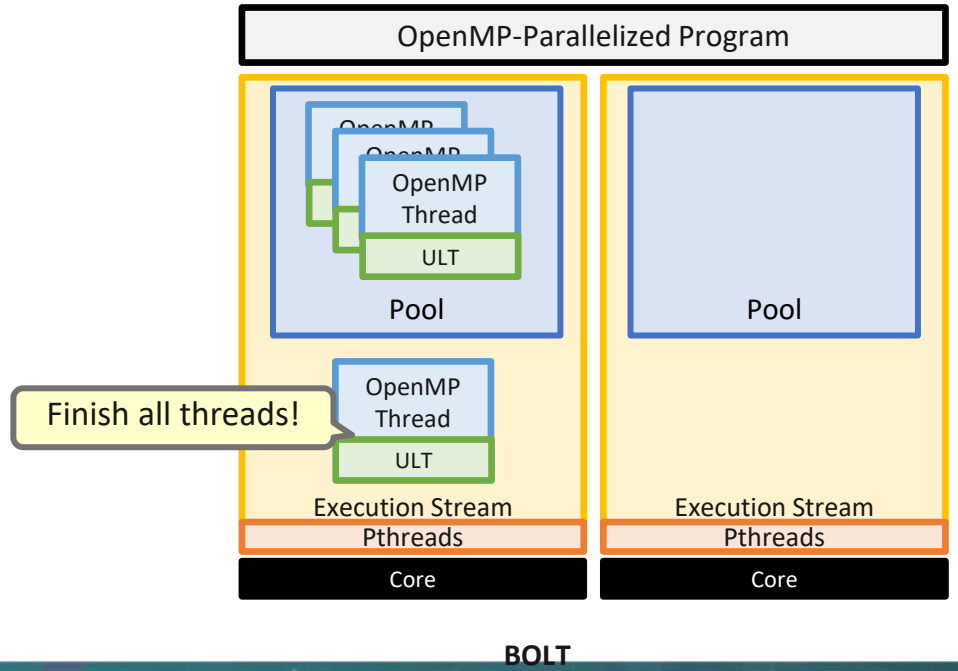


BOLT

How does BOLT Work?

- Parallel region (e.g., `#omp parallel for`) creates OpenMP threads over ULTs.
- Work stealing can distribute work.

```
#pragma omp parallel for num_threads(4)
for (i = 0; i < 4; i++)
  kernel(data[i], i);
```



  - OpenMP task works similarly, but this talk omits the details.

Lightweight Threads for OpenMP+OpenMP

Lightweight Threads for OpenMP + OpenMP

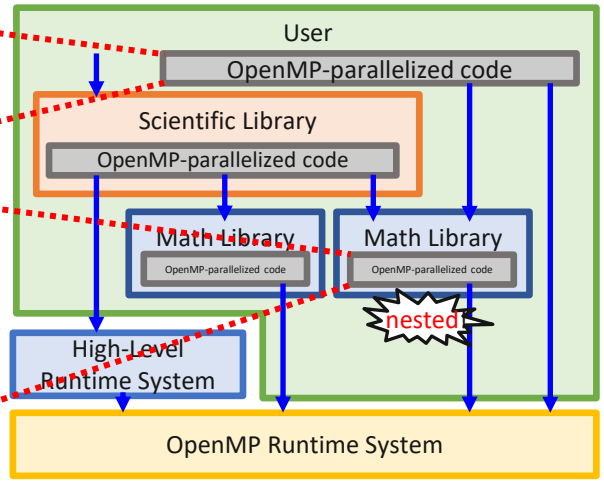
- Case of oversubscriptions: nested parallel regions

```
#pragma omp parallel for
for (i = 0; i < n; i++)
  dgemv(matrix[n], ...);
```

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

nested

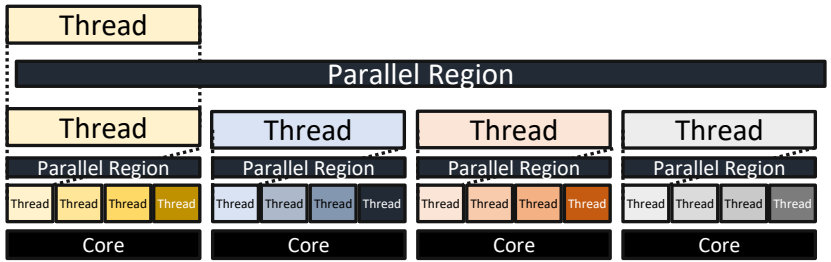
Code Example



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

```
#pragma omp parallel for
for (i = 0; i < n; i++)
  dgemm(matrix[n], ...);
```

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

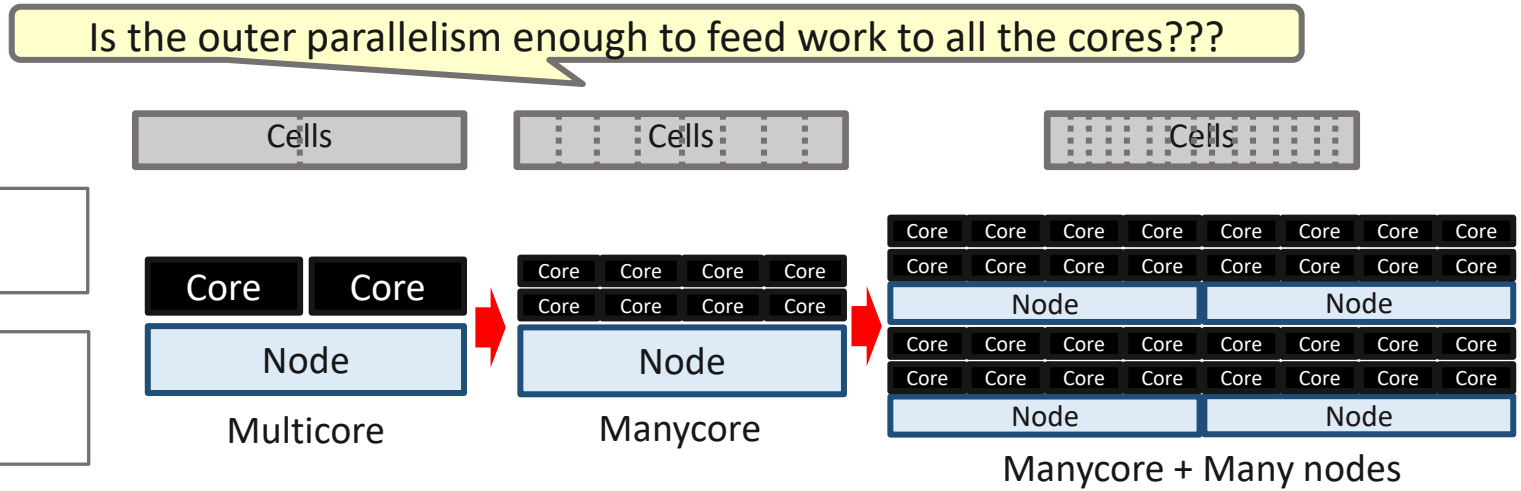


Can We Just Disable Nested Parallelism?

- 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

```
#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++);
```



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[*].
- Investigated two directions to address it:

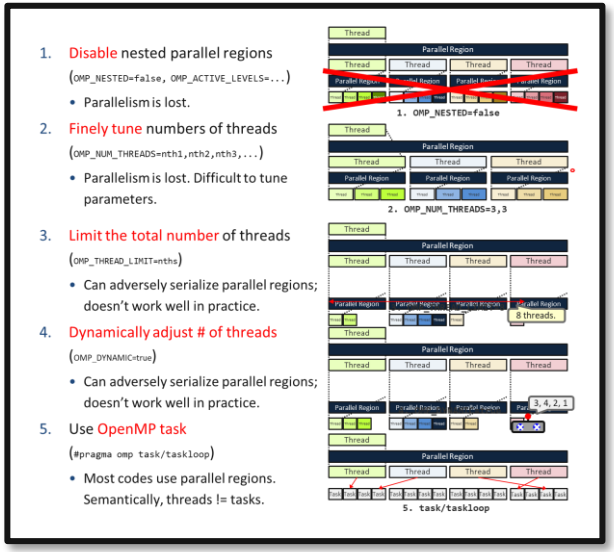
  1. Use several workarounds implied in the OpenMP specification.

  => Inefficient/unfeasible if users do not know parallelism in other software stacks.

  2. Instead of OS-level threads, use lightweight threads as OpenMP threads

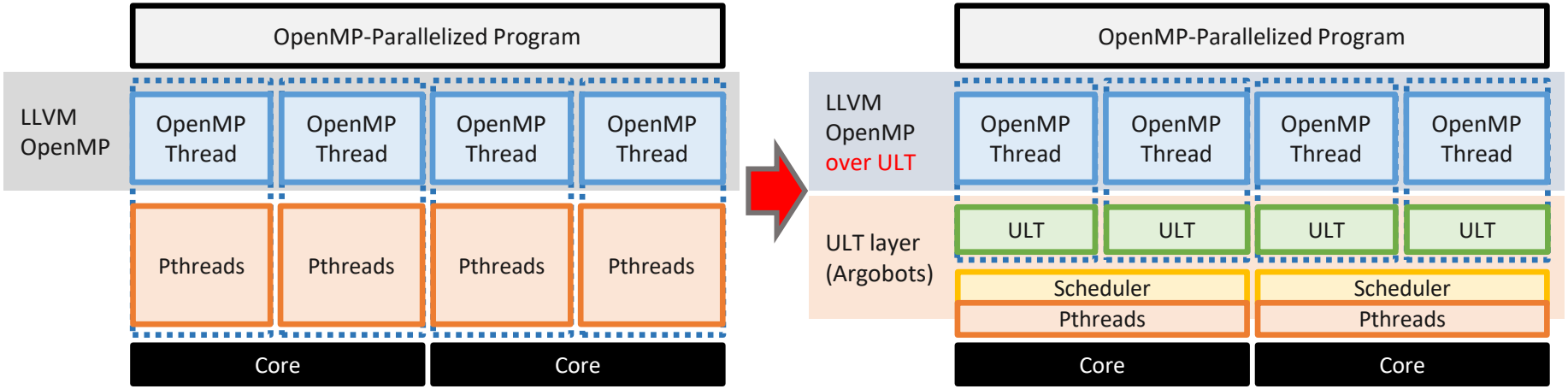
  => It does not perform well if parallel regions are not nested (i.e., flat).
    - It does not perform well even when parallel regions are nested.

=> 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.

Using ULTs in LLVM OpenMP is Easy



LLVM OpenMP 10.0

LLVM OpenMP 10.0 over ULT (= BOLT baseline)

- Replacing a Pthreads layer with that of Argobots is easy
- Does the "baseline BOLT" perform well?

ULT-based OpenMP is not New.

- Use ULTs to avoid oversubscriptions of OS-level threads.
- 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] …

[1] Marc et al., NanosCompiler: Supporting Flexible Multilevel Parallelism Exploitation in OpenMP. 2000
[2] Tanaka et al., Performance Evaluation of OpenMP Applications with Nested Parallelism. 2000
[3] Hadjidoukas et al., Support and Efficiency of Nested Parallelism in OpenMP Implementations. 2008
[4] Pérache et al., MPC: A Unified Parallel Runtime for Clusters of NUMA Machines. 2008
[5] Broquedis et al., ForestGOMP: An Efficient OpenMP Environment for NUMA Architectures. 2010
[6] Duran et al., A Proposal for Programming Heterogeneous Multi-Core Architectures. 2011
[7] Broquedis et al., libKOMP, an Efficient OpenMP Runtime System for Both Fork-Join and Data Flow Paradigms. 2012

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

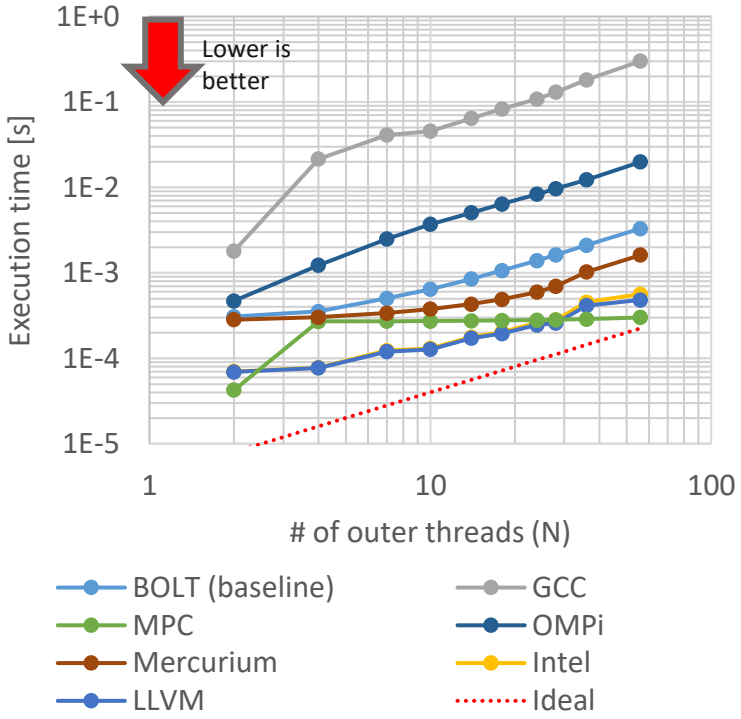
For apples-to-apples comparison, we will focus on the ULT-based LLVM OpenMP.

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



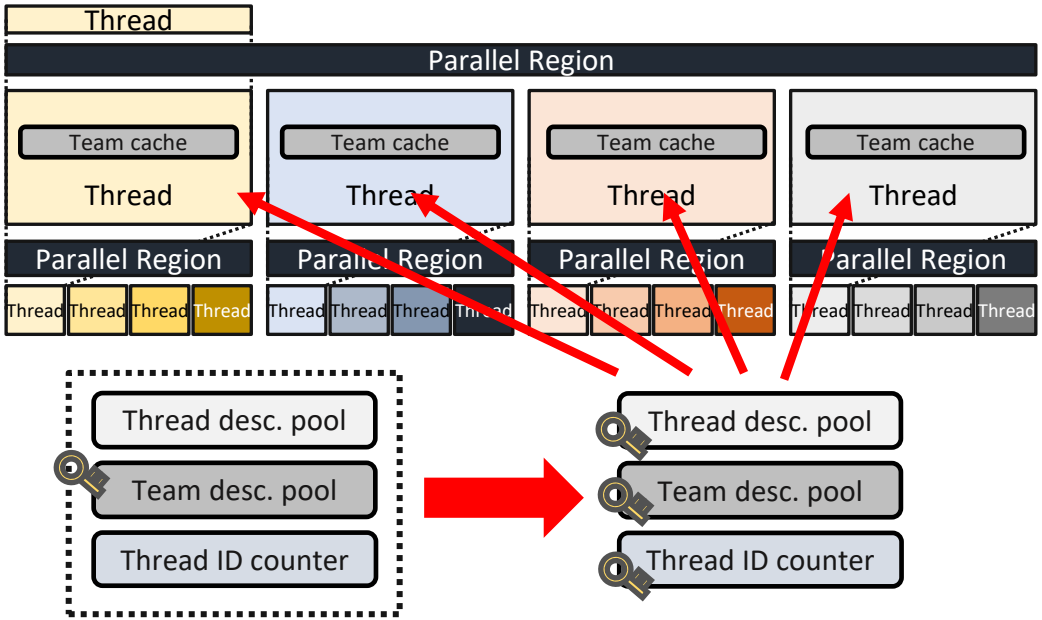
Popular Pthreads-based OpenMP

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

Solve Scalability Bottlenecks (1/2)

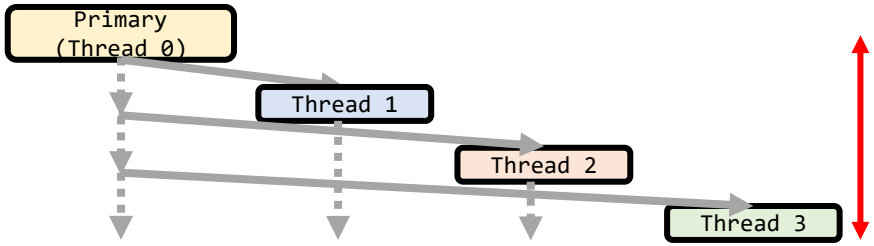
- Resource management optimizations
  1. Divides a large critical section protecting all threading resources.
     - This cost is negligible with Pthreads.
  2. Enable multi-level caching of parallel regions
     - Called “nested hot teams” in LLVM OpenMP.



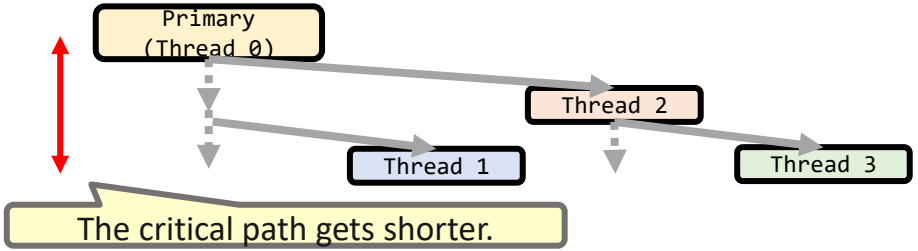
Solve Scalability Bottlenecks (2/2)

- Thread creation optimizations

3. Binary creation of OpenMP threads.



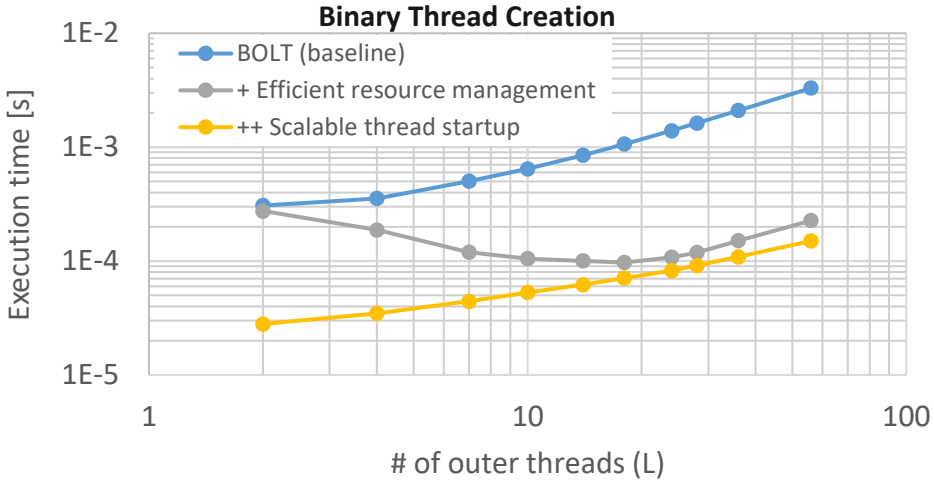
Serial Thread Creation (default LLVM OpenMP)



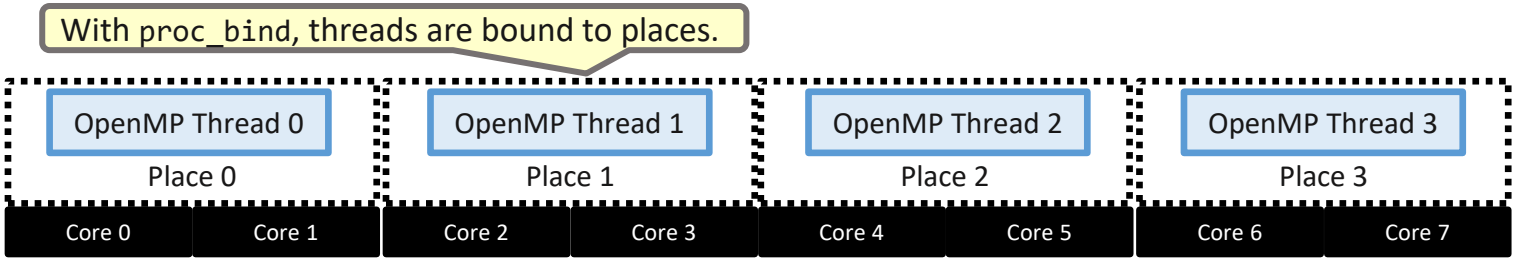

```
// Run on a 56-core Skylake server
#pragma omp parallel for num_threads(L)
for (int i = 0; i < L; i++)
  #pragma omp parallel for num_threads(56)
  for (int j = 0; j < 56; j++)
    no_comp();
```

Nested Parallel Regions (no computation)

No computation to measure the pure overheads.



Affinity: How to Implement Affinity for ULTs



- OpenMP 4.0 introduced *place* and *prod_bind* for affinity.
  - OS-level thread-based libraries use CPU masks.
- BOLT (baseline) ignored affinity (still standard compliant).
- However, affinity should be useful to

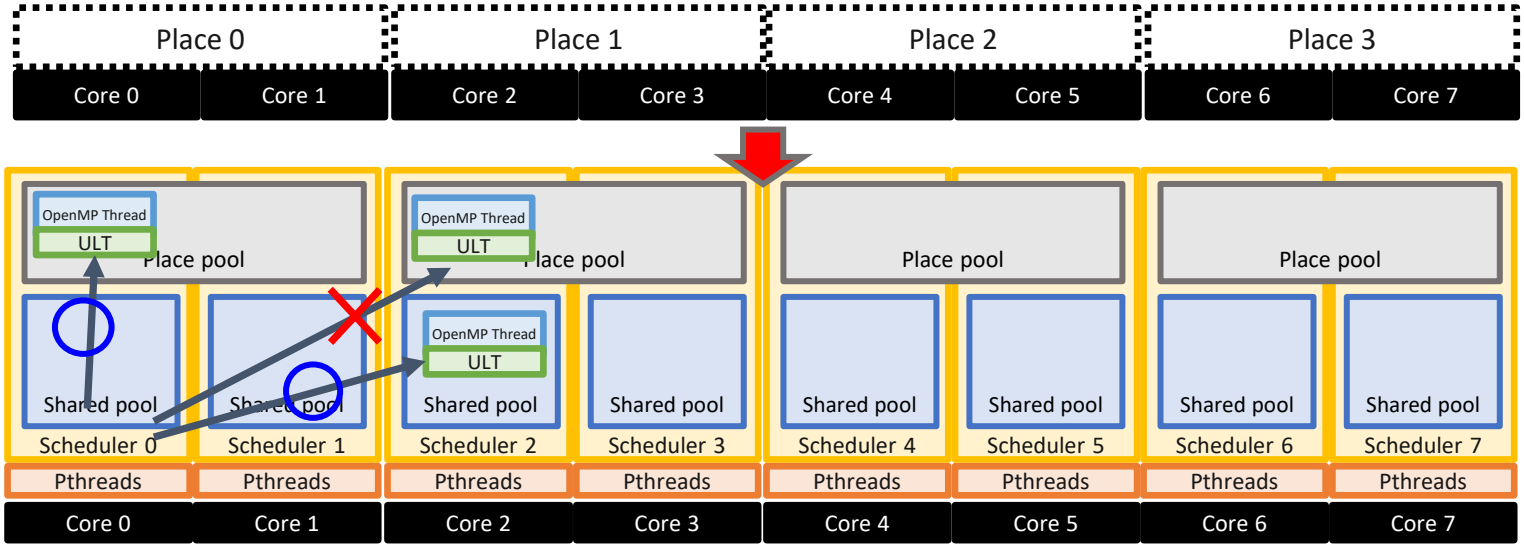
  1. improve locality and 2. reduce pool contentions.
  - Note: ULT runtimes use shared pools + random work stealing.
- *How to implement place over ULTs*?

```
// 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 Pool

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

```
// 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);
```



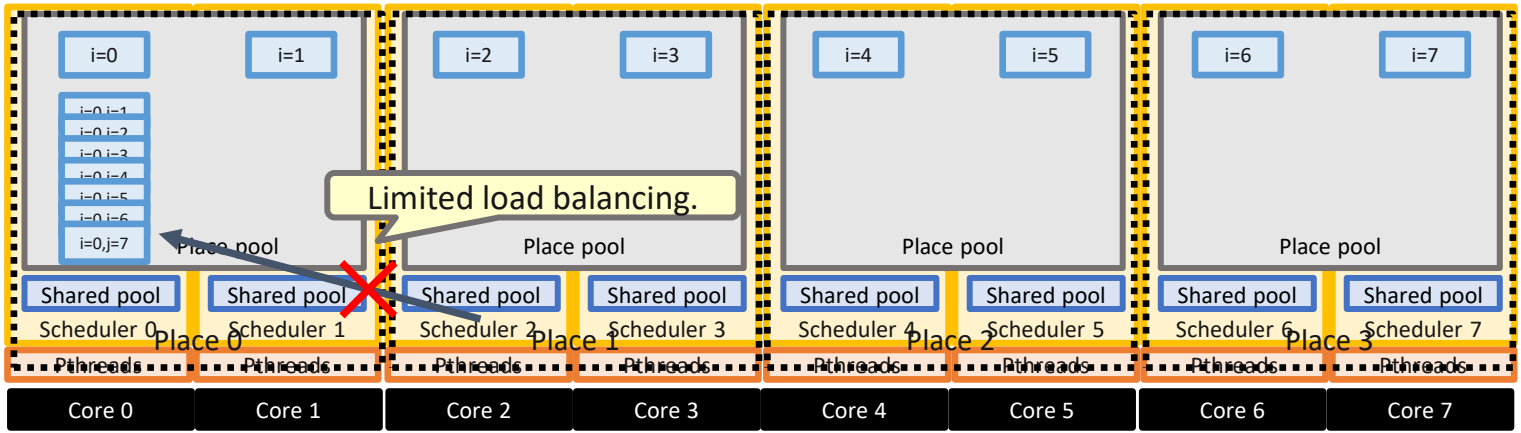
- Problem: OpenMP affinity setting is too deterministic.

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.

The OpenMP specification writes so.

```
// 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 random work stealing.

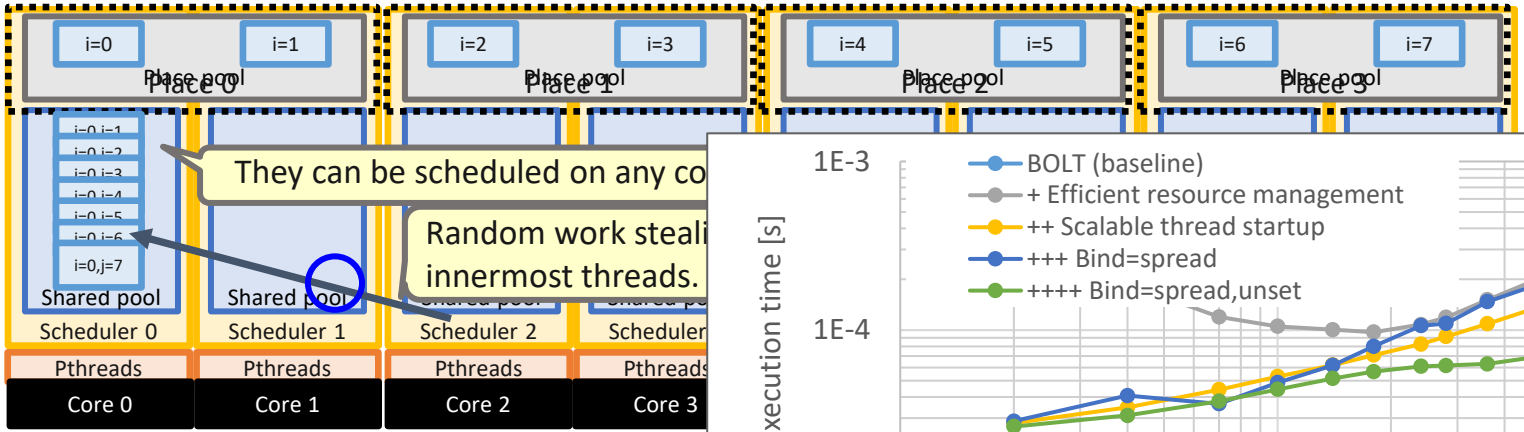
Proposed New PROC_BIND: “unset”

`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);
```

```
// OMP_PLACES={0,1},{2,3},{4,5},{6,7}
// OMP_PROC_BIND=spread,unset
#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);
```

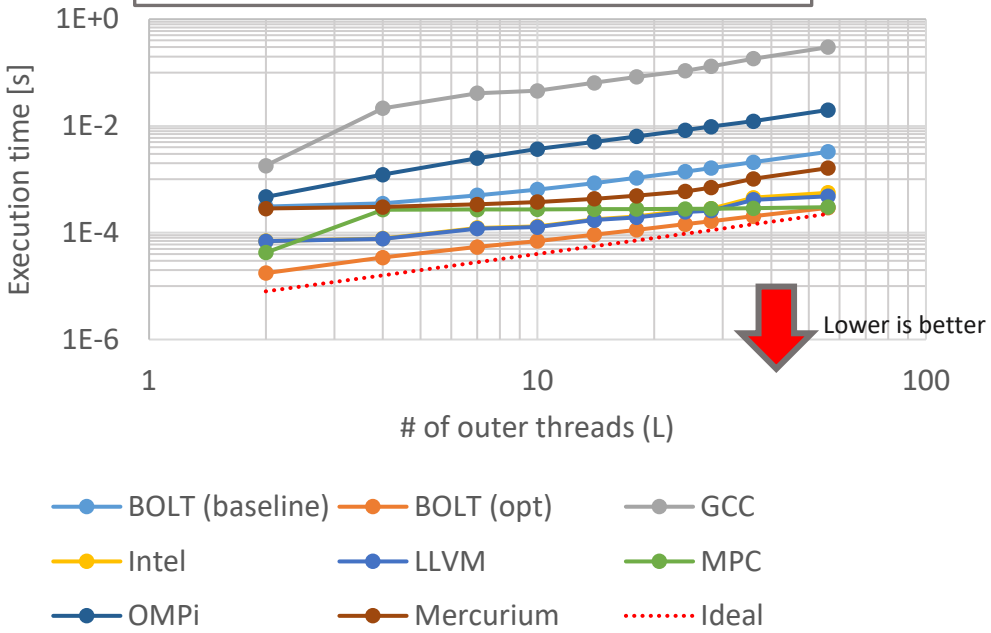


- This scheduling flexibility gives higher performance.

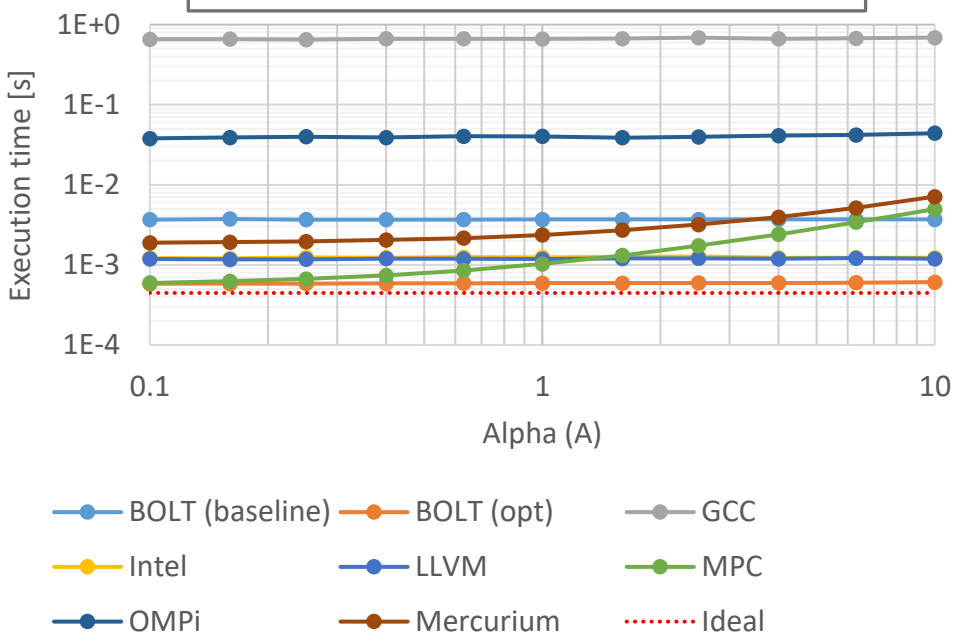
Microbenchmarks: Doubly Nested Loops

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


```
// 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_thLreads(28)
  for (int j = 0; j < 28; j++)
    comp_20000_cycles(i, j);
}
```



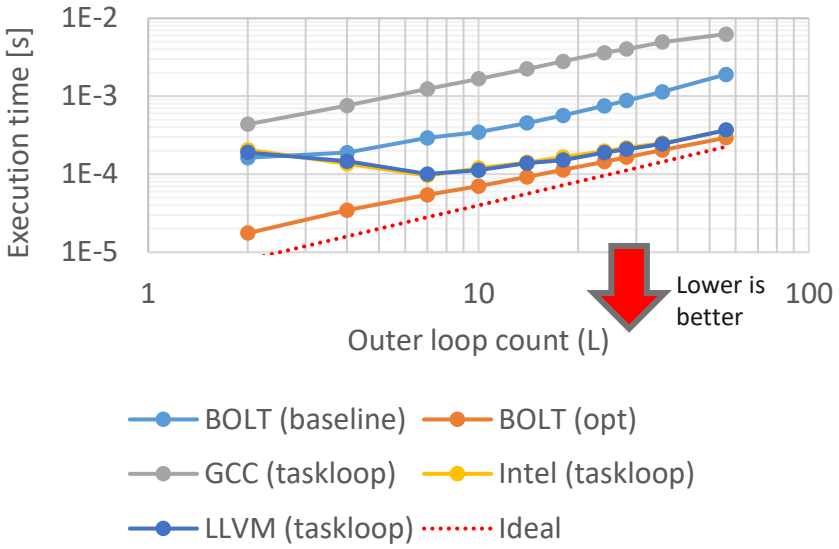

```
// Run on a 56-core Skylake server
#pragma omp parallel for num_threads(56)
for (int i = 0; i < 56; i++) {
  int work_cycles = get_work(i, alpha);
  #pragma omp parallel for num_threads(56)
  for (int j = 0; j < 56; j++)
    comp_cycles(i, j, work_cycles);}
```



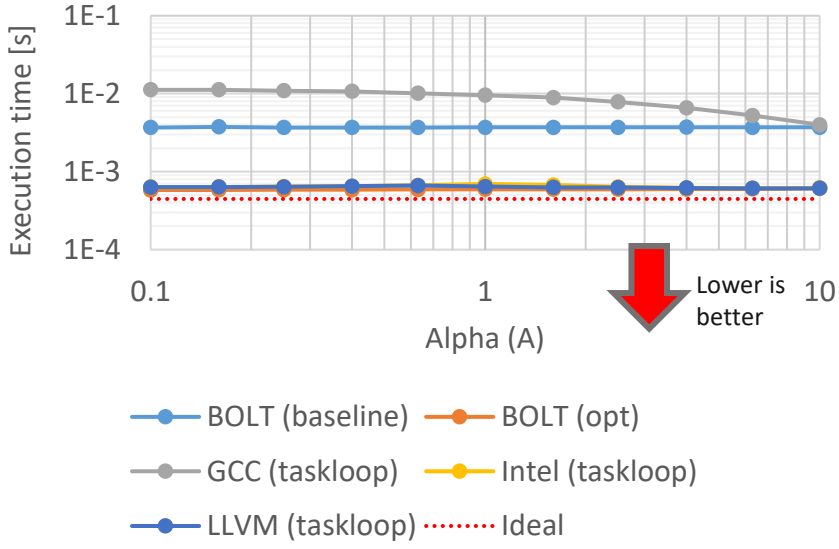
(Ideal): lower bound under perfect scalability.

OpenMP Threads vs. OpenMP Tasks

```
// 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 taskloop grainsize(1)
  for (int j = 0; j < 56; j++)
    comp_cycles(i, j, work_cycles);}
```

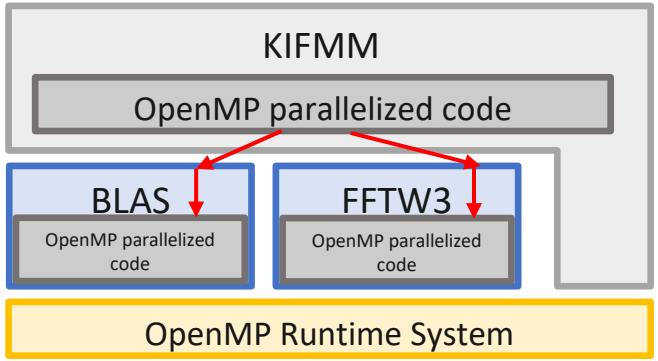


- Parallel regions of BOLT are as fast as `taskloop`!

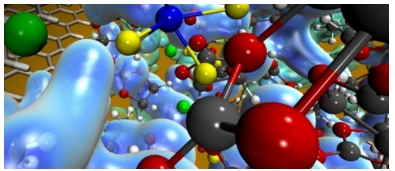
Evaluation of BOLT with Two Applications

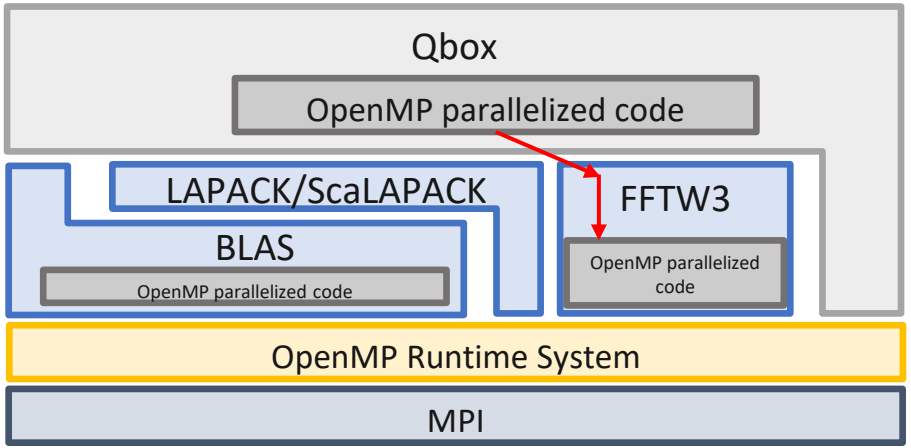
- # of threads for outer loops is usually set to # of cores.
  - i.e., if not nested, oversubscription does not happen.
- If many layers are OpenMP parallelized, nesting can unintentionally happen.
- Two showcases:

1. KIFMM



2. Qbox

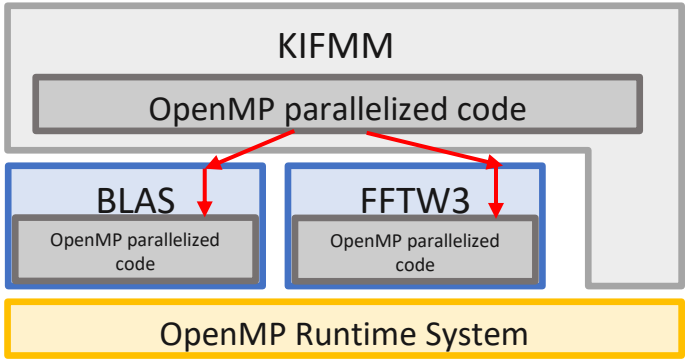




Evaluation: KIFMM

- 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.

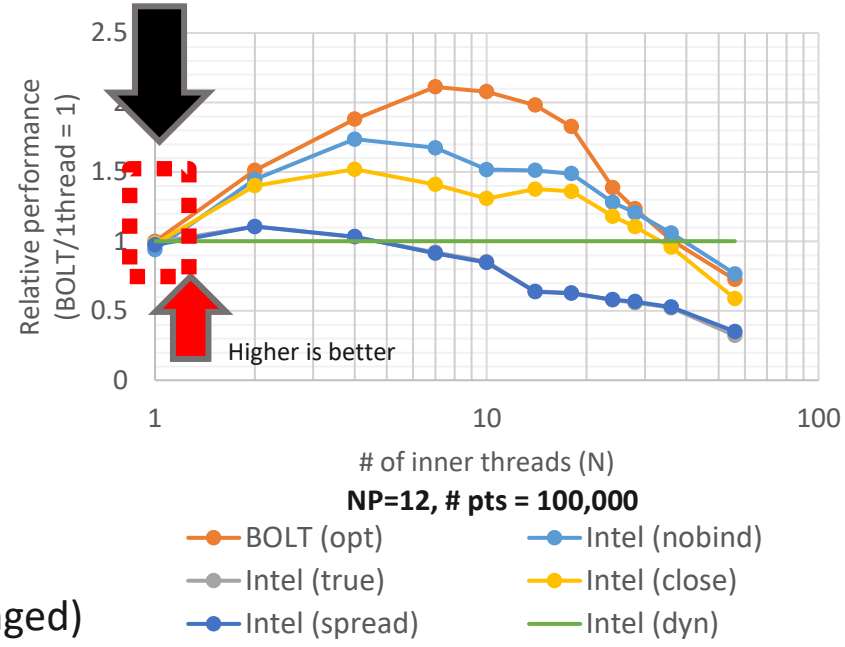
```
for (int i = 0; i < max_levels; i++)
  #pragma omp parallel for
  for (int j = 0; j < nodecounts[i]; j++) {
    [...];
    dgemv(...); // dgemv() creates a parallel region.
  }
```



[*] A. Chandramowlishwaran et al., "Brief Announcement: Towards a Communication Optimal Fast Multipole Method and Its Implications at Exascale", SPAA '12, 2012

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(...);
```

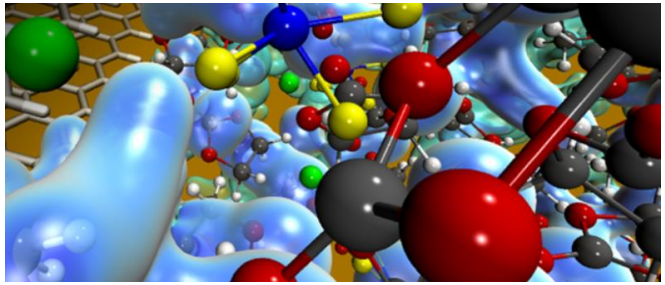


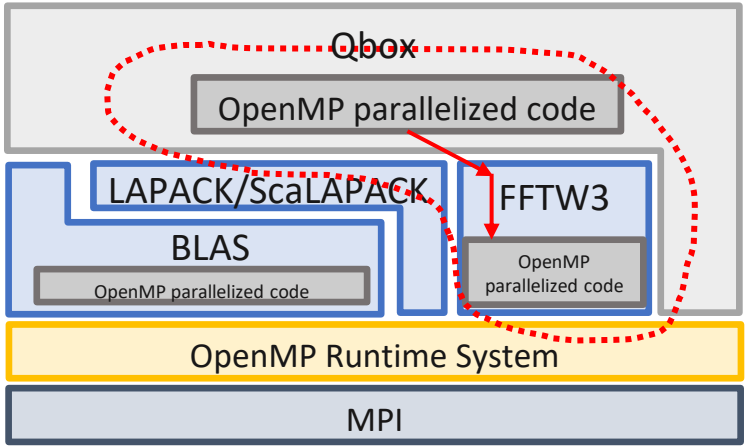
- Experiments on Skylake 56 cores.
  - # of threads for the outer parallel region = 56
  - # of threads for the inner parallel region = N (changed)
- Two important results:
  - N=1 (flat): performance is almost the same.
  - N>1 (nested): BOLT further boosts performance.

Different Intel OpenMP configurations:
nobind: OMP_PROC_BIND=false,
true, close, spread : OMP_PROC_BIND= {true, close, spread}
dyn: MKL_DYNAMIC=true
Note that other parameters are hand tuned (see the paper).

Evaluation: FFT in Qbox

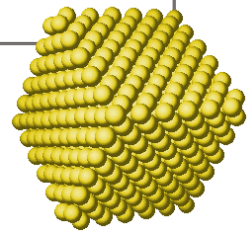
- Qbox[*]: first-principles molecular dynamics code.
- We focus on the FFT computation part.






```
// 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(...);
```



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

[*] F. Gygi, "Architecture of Qbox: A scalable first-principles molecular dynamics code," IBM Journal of Research and Development, vol. 52, no. 1.2, pp. 137–144, Jan. 2008.

Performance: FFT in Qbox

```
// 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(...);
```

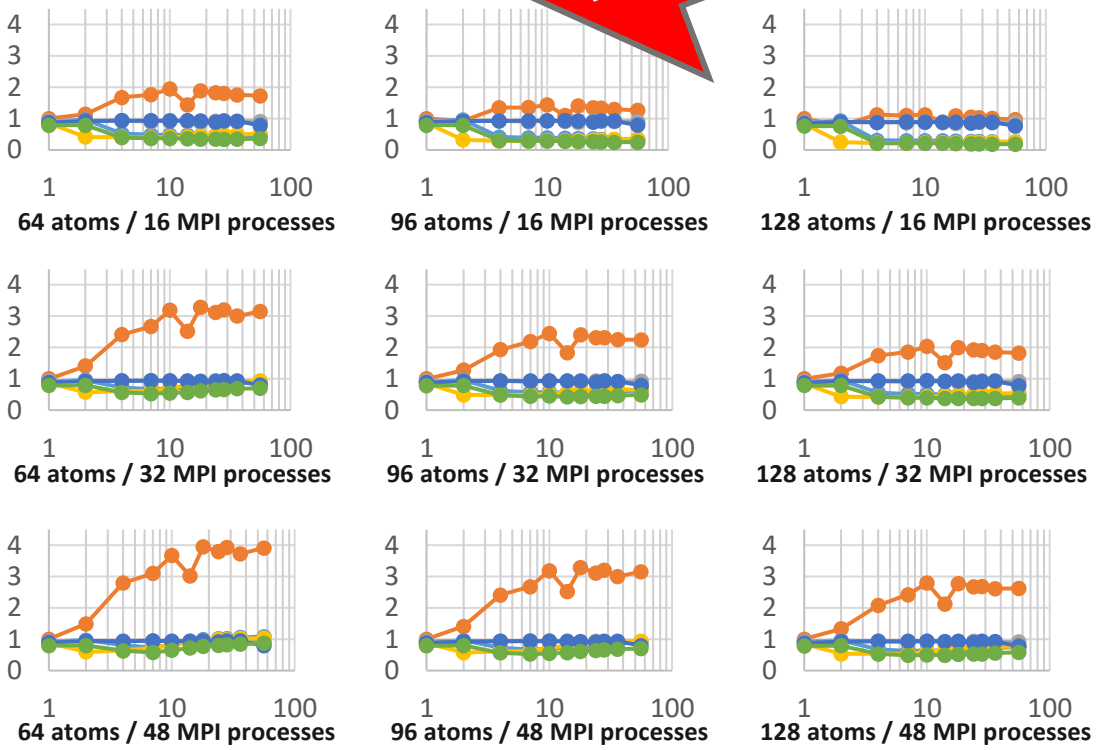
BOLT (opt) — Intel (nobind)
Intel (true) — Intel (close)
Intel (spread) — Intel (dyn)

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

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

Experiments on KNL 7230 64 cores.
threads for the outer regions = 64
threads for the inner regions = N (changed)

More beneficial for nested parallel regions.
=> Strong scaling



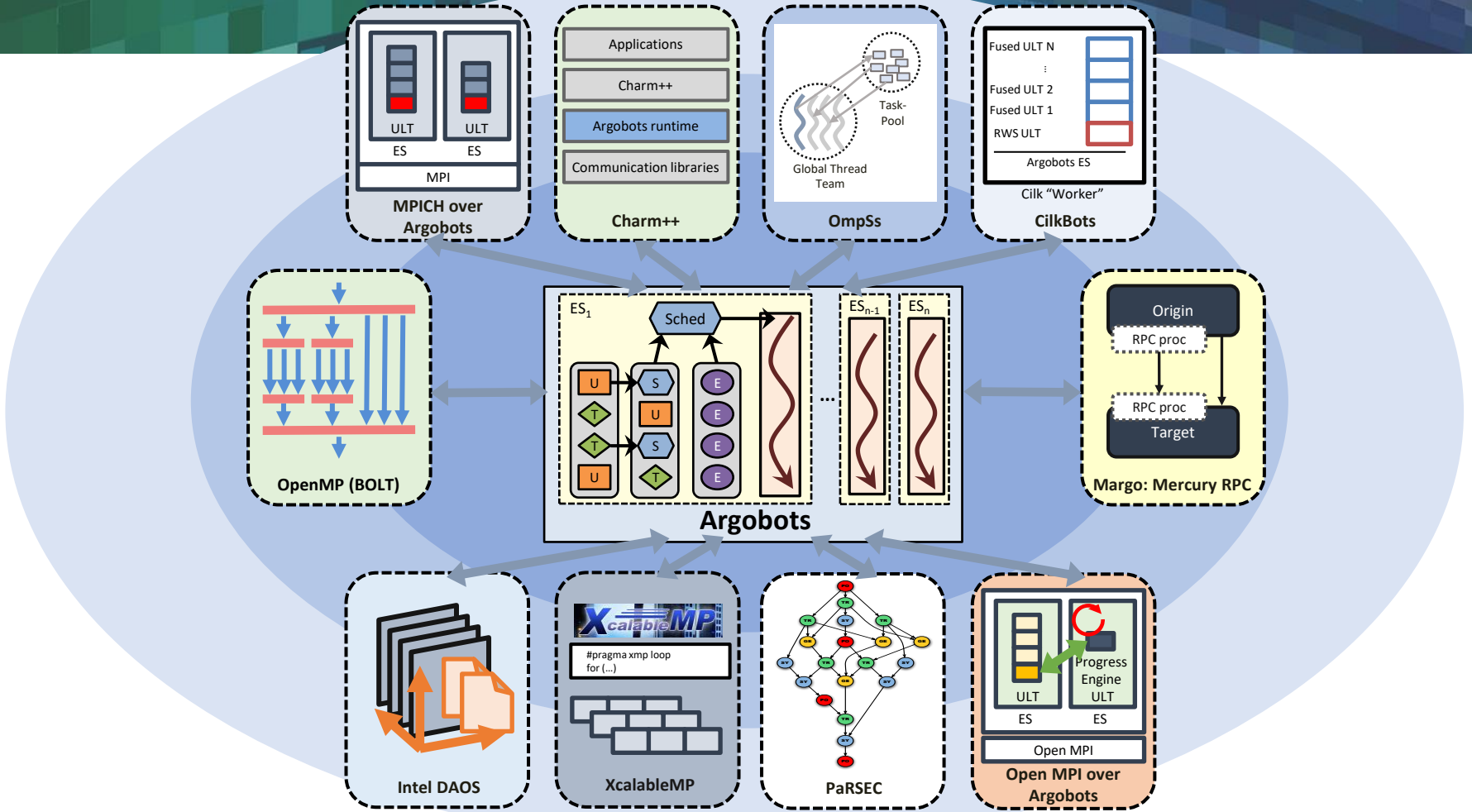
X axis: # of inner threads (N)
Y axis: relative performance (BOLT + N=1: 1.0)

Higher is better

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

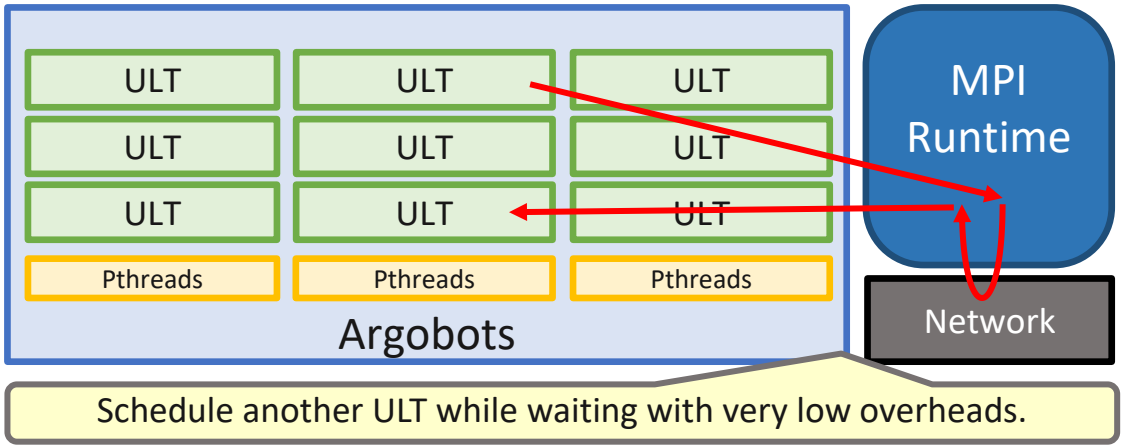
Lightweight Threads for OpenMP+MPI

Ecosystem of Argobots



High Interoperability with MPI [*]

- MPI+BOLT can improve performance via Argobots
  - ULTs are flexibly scheduled without kernel intervention.
    - Better communication and computation overlapping
    - Latency hiding of blocking operations.
- Current work
  - Open MPI + Argobots
  - MPICH + Argobots

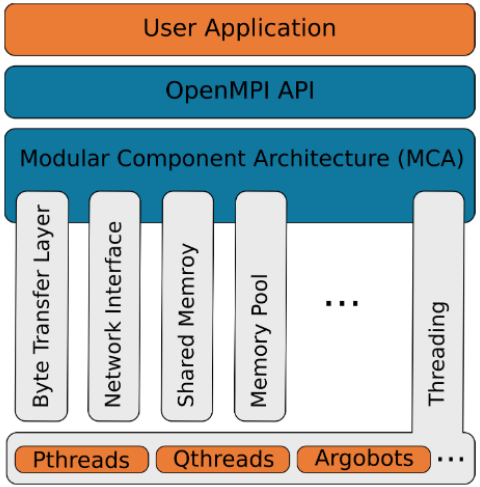


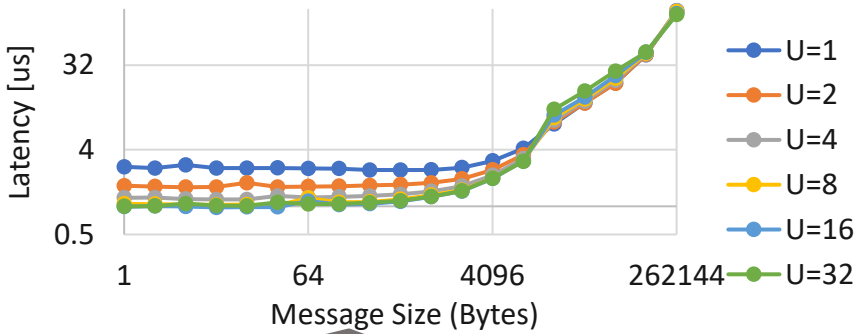
[*] H. Lu et al. "MPI+ULT: Overlapping Communication and Computation with User-Level Threads", HPCC '15, 2015

BOLT/Argobots + MPI

- MPICH + BOLT/Argobots is officially supported.

Stable! Passes all the major multithreaded tests; as good as Pthreads!
(We test it weekly)





Point-to-point latency with 36 Haswell cores and Mellanox FDR (U = # of ULTs per ES), MPICH

- Initial support for Open MPI + BOLT/Argobots has been merged to the release branch.
  - In collaboration with SNL and LANL Qthreads/Open MPI teams.

It now covers about 80% of the MPICH multithreaded tests. It is under active development.

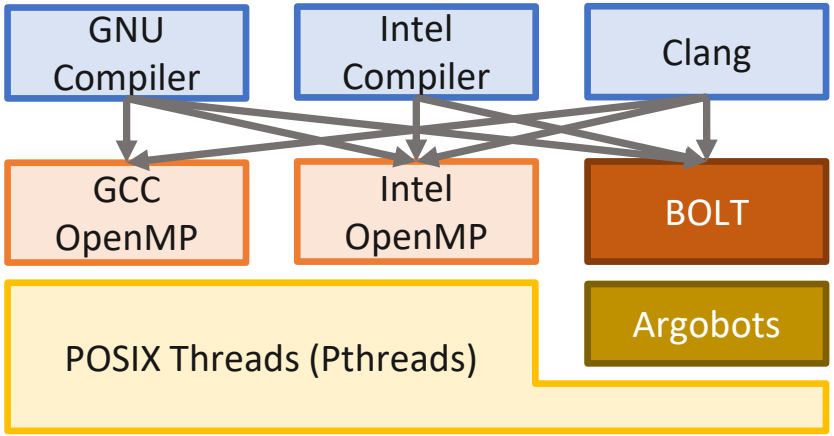
Conclusions

High ABI Compatibility

- 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.

```
# app is compiled for GCC or Intel OpenMP
# run app over a default OpenMP library
./app arg1 arg2 ...
```

```
# run app over BOLT
LD_PRELOAD=$BOLT_LIB_PATH ./app arg1 arg2 ...
```



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

Support of the Latest Features

- BOLT supports OpenMP 5.0 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 itself 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 and task cancellation.

How to Install BOLT?

1. Use Spack

```
$ spack install bolt   # only BOLT
$ spack install sollve # all the SOLLVE components
```

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

```
$ 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 ../ -DCMAKE_INSTALL_PREFIX=BOLT_INSTALL_DIR \
      -DCMAKE_BUILD_TYPE=Release -DLIBOMP_USE_ARGOBOTS=on
$ make -j install
```

How to Use BOLT?

- ***No recompilation needed. Just change the OpenMP runtime library.***
- [Recommended] Set LD_LIRBARY_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)
```

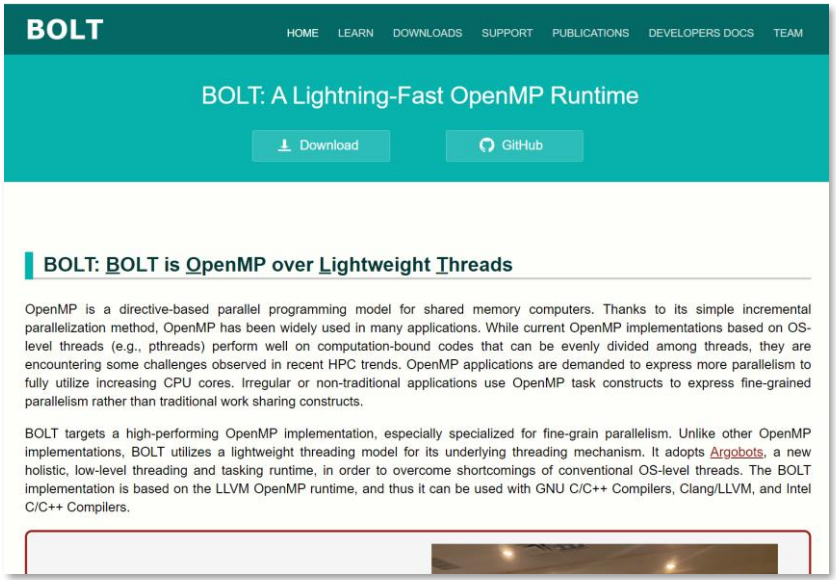
If you cannot find it, LD_LIBRARY_PATH does not work. It often happens if you use GCC as a frontend.

- [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
```

BOLT: Lightweight OpenMP Runtime

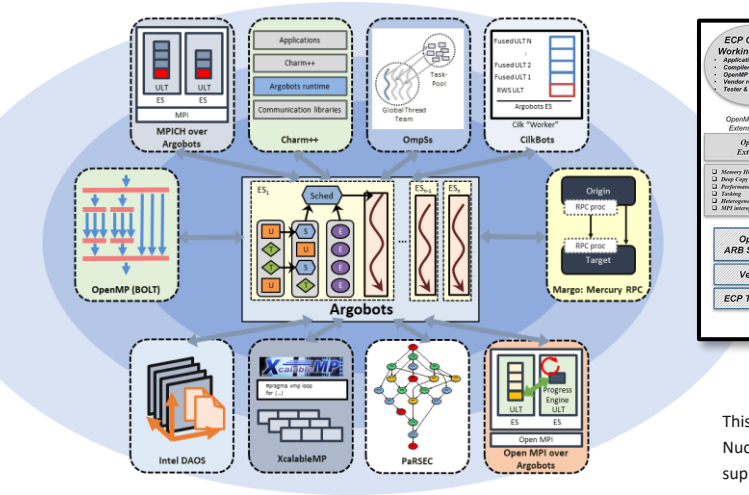
- BOLT: a lightweight OpenMP runtime system based on LLVM OpenMP for efficient threading in OpenMP + X.
  1. Extremely lightweight OpenMP threads (aka. `omp parallel for`) that can efficiently handle nested parallelism.
  2. High interoperability with MPI.
  - Please visit us!
    - https://www.bolt-omp.org/
    - https://github.com/pmodels/bolt
    - or google “BOLT OpenMP”

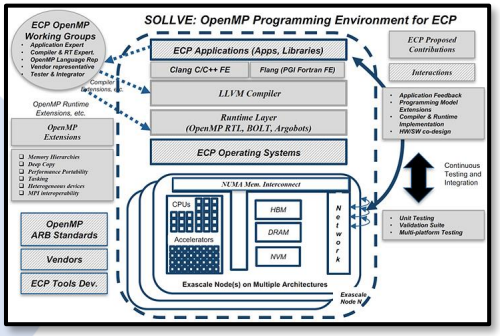


Thank You for Listening!

```
$ spack install bolt   # only BOLT
$ spack install sollve # all the SOLLVE components
```

- BOLT-Related Publications:
  - H. Lu et al. "MPI+ULT: Overlapping Communication and Computation with User-Level Threads", HPCC '15, 2015
  - A. Castelló et al. A Review of Lightweight Thread Approaches for High Performance Computing, CLUSTER '16, 2016
  - S. Seo et al. "Argobots: A Lightweight Low-Level Threading and Tasking Framework", TPDS, 2018
  - S. Iwasaki et al., "Lessons Learned from Analyzing Dynamic Promotion for User-Level Threading", SC '18, 2018
  - S. Iwasaki et al., “BOLT: Optimizing OpenMP Parallel Regions with User-Level Threads", PACT ‘19, 2019 (**Best Paper Award**)
  - S. Iwasaki et al., “Analyzing the Performance Trade-Off in Implementing User-Level Threads", TPDS, 2020





https://www.anl.gov/mcs/article/researchers-win-best-paper-award-at-pact19

- BOLT is part of the ECP SOLLVE project.
  - https://www.bnl.gov/compsci/projects/SOLLVE/

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.

For the OpenMP specification, tutorials, forum, reference guides, and links to other resources, visit **www.openmp.org**