Accommodating Thread-Level Heterogeneity in Coupled Parallel Applications

Samuel K. Gutiérrez, Kei Davis, Dorian C. Arnold, Randal S. Baker
Robert W. Robey, Patrick McCormick, Daniel Holladay, Jon A. Dahl
R. Joe Zerr, Florian Weik, and Christoph Junghans

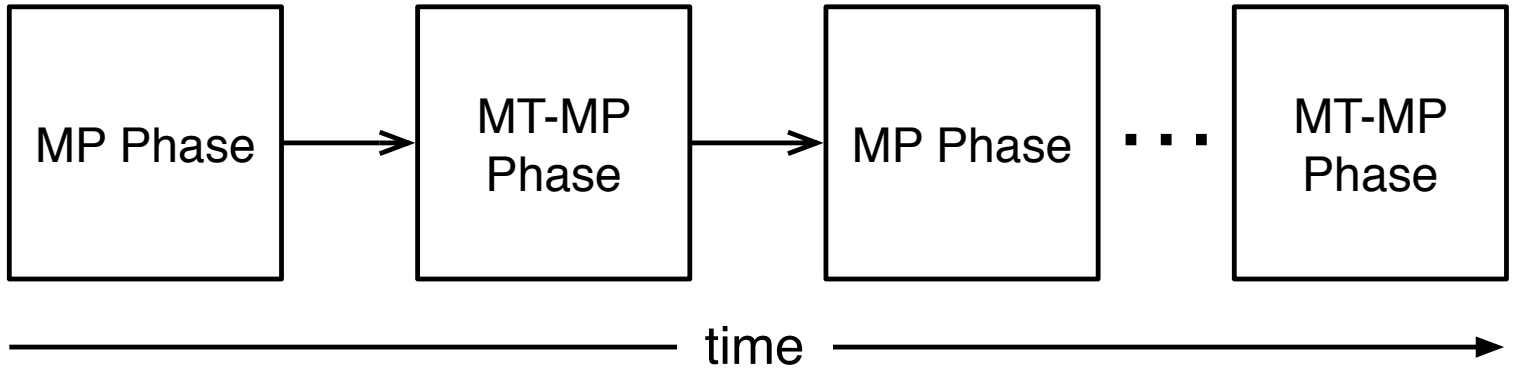
Programming Models Team, Applied Computer Science, LANL
Scalable Systems Lab, Department of Computer Science, UNM

Los Alamos NATIONAL LABORATORY EST. 1943

UNM

New, Hybrid Programming Models are Emerging

- Multi-threaded message-passing (MT-MP) applications becoming more popular
- MP + MT-MP commonplace in *coupled applications*
  - MP and MT-MP libraries linked together with interleaved execution

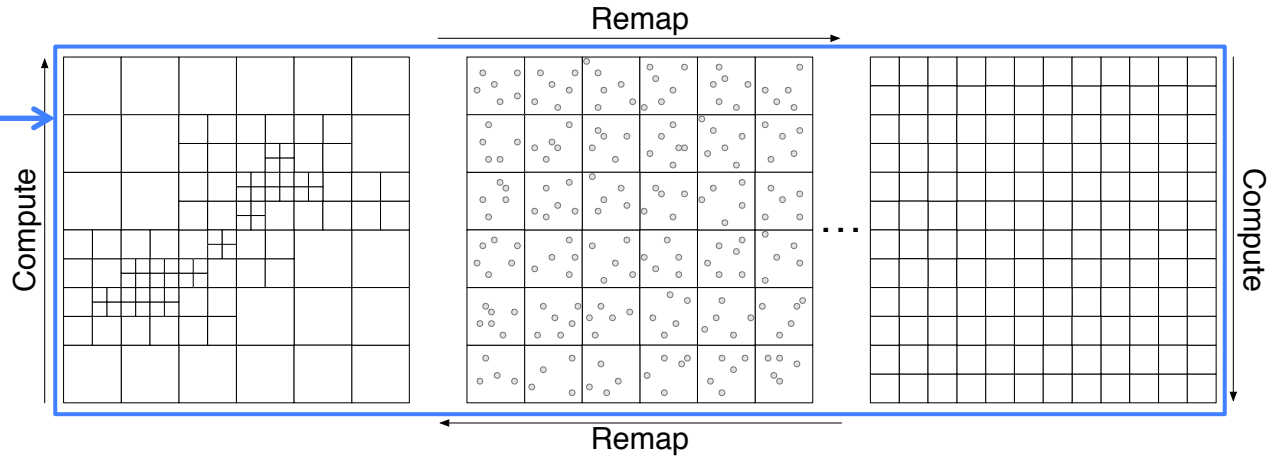


Where does this Type of Coupling Occur?

Parallel and distributed multi-physics applications

- Crucial in science and engineering
- Often interdisciplinary effort
  - Built by integration (or *coupling*) of independently developed and tuned software libraries linked into a single application

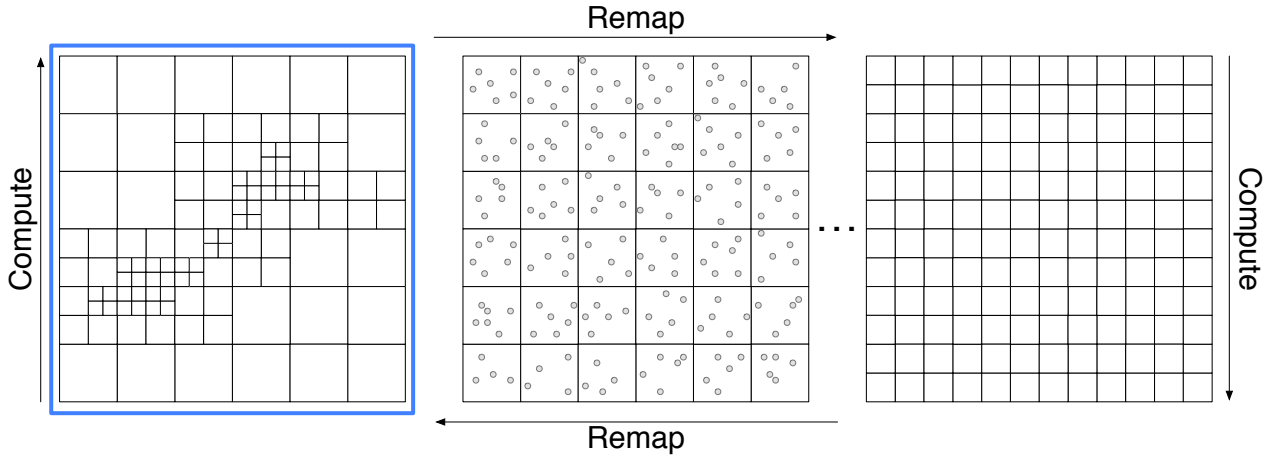
Phases of single, coupled application



Where does this Type of Coupling Occur?

Parallel and distributed multi-physics applications

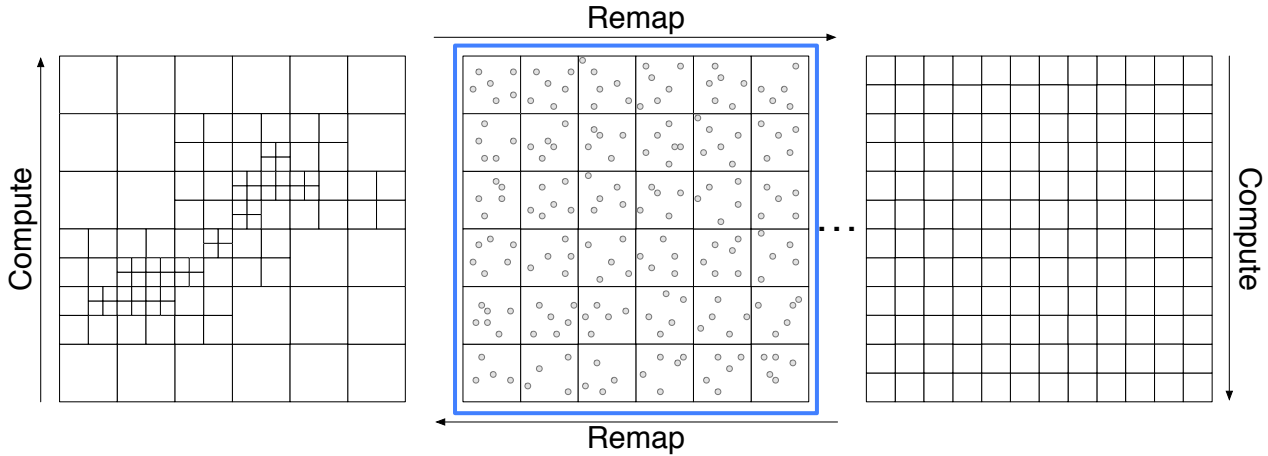
- Crucial in science and engineering
- Often interdisciplinary effort
  - Built by integration (or *coupling*) of independently developed and tuned software libraries linked into a single application



Where does this Type of Coupling Occur?

Parallel and distributed multi-physics applications

- Crucial in science and engineering
- Often interdisciplinary effort
  - Built by integration (or *coupling*) of independently developed and tuned software libraries linked into a single application



Challenges of Coupling Disparate Libraries

Each library should execute based on its design and tuning

- Each has own optimal *runtime configuration*
  - E.g, number and placement of *tasks* (processes and threads)
  - Poor library configuration → poor library performance → poor application performance

- Static parameters – found manually, heuristically and offline

- *Configuration conflicts* arise when an optimal library configurations conflict with each other

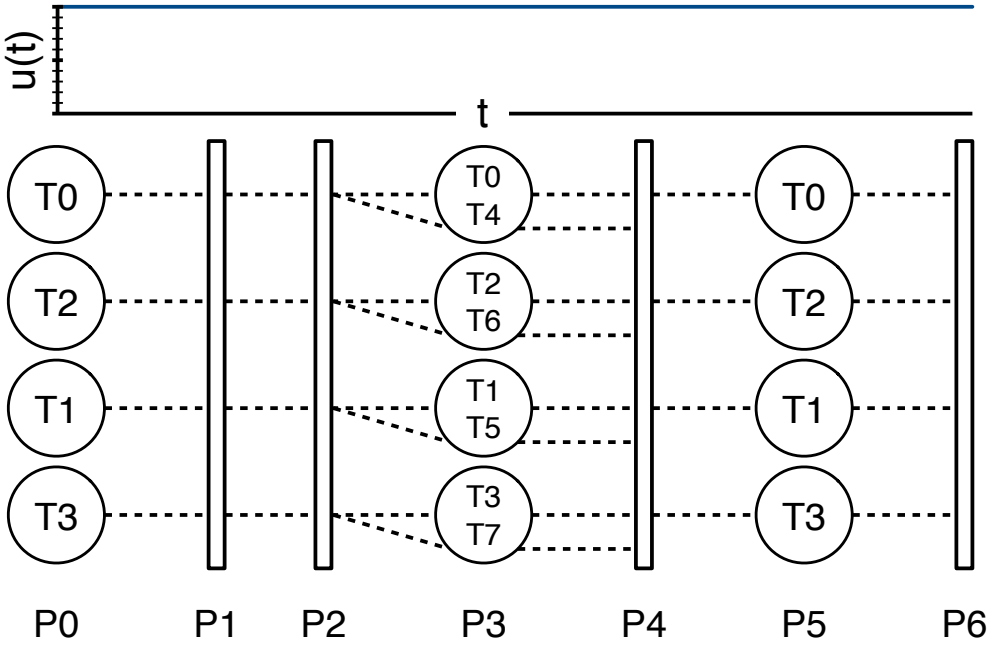
State of the Practice

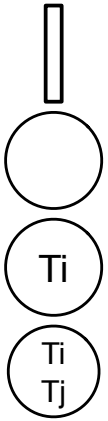
- Task-to-hardware bindings can improve performance

- Parallel launchers (e.g., srun, orterun) only support static binding and placement
  - Launch-time configuration persists for entire execution

- Two basic, static configuration options:
  - *Over-subscription* or *under-subscription*

Today's Static Configurations: Over-Subscription





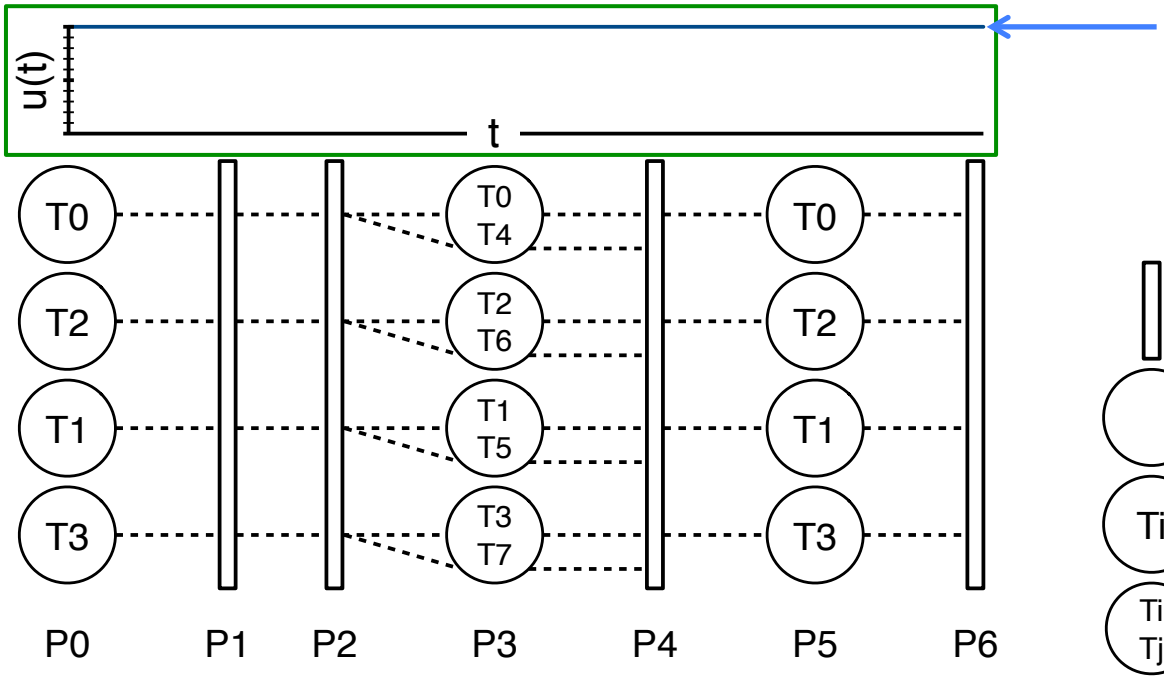
Parallel compute phase

Vacant processing element (PE)

PE occupied by task Ti

PE occupied by tasks Ti and Tj

Today's Static Configurations: Over-Subscription



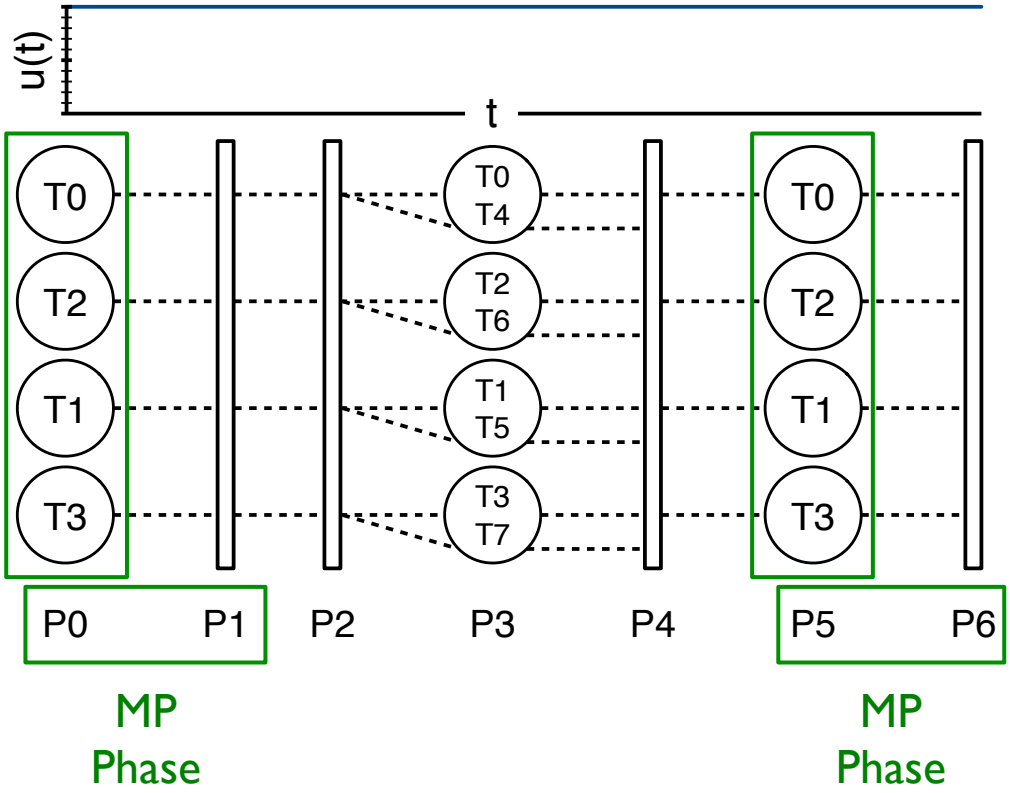
Parallel compute phase

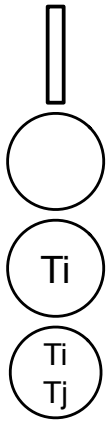
Vacant processing element (PE)

PE occupied by task Ti

PE occupied by tasks Ti and Tj

Today's Static Configurations: Over-Subscription





Parallel compute phase

Vacant processing element (PE)

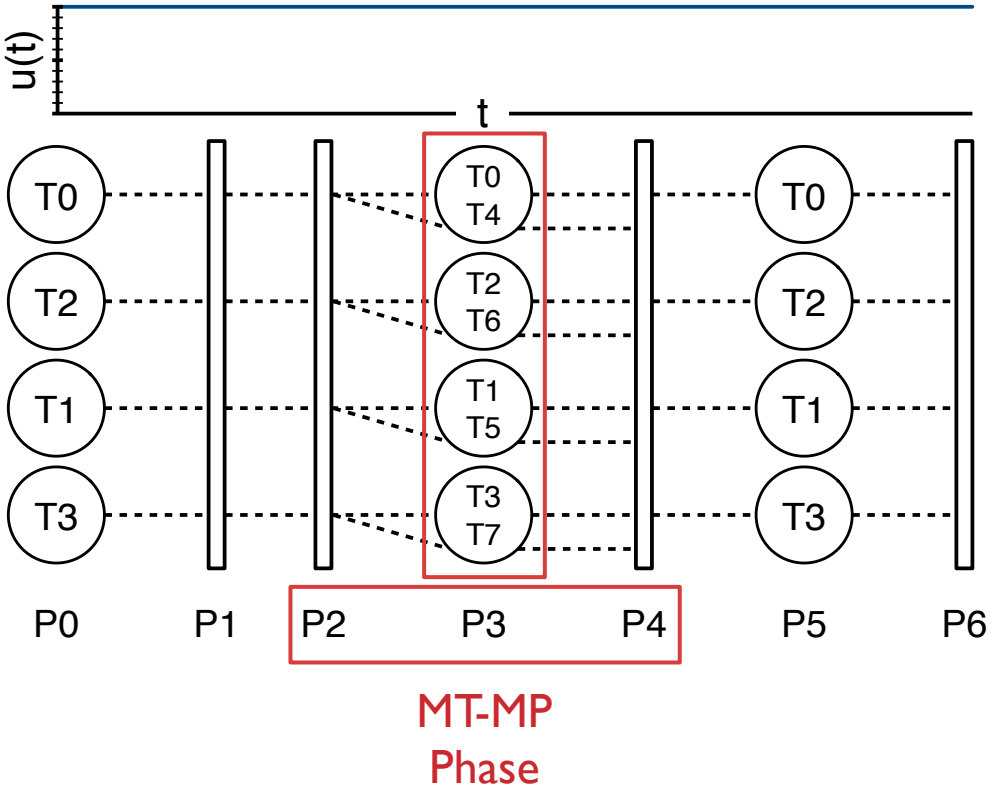
Ti

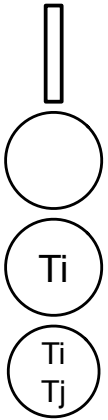
PE occupied by task Ti

Ti
Tj

PE occupied by tasks Ti and Tj

Today's Static Configurations: Over-Subscription





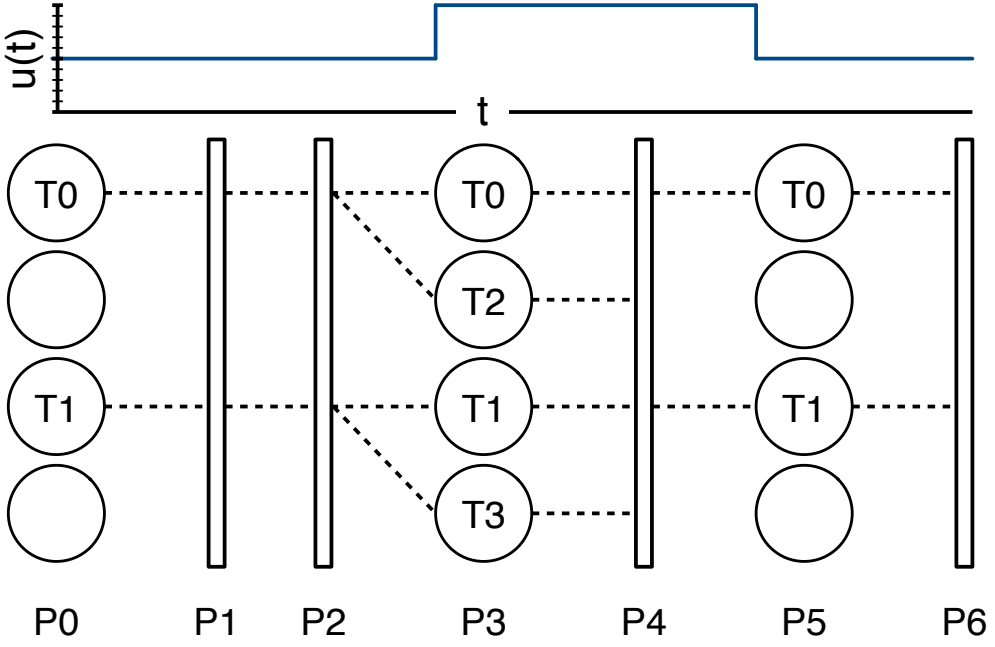
Parallel compute phase

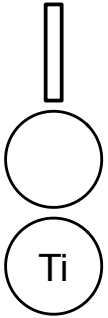
Vacant processing element (PE)

PE occupied by task Ti

PE occupied by tasks Ti and Tj

Today's Static Configurations: Under-Subscription



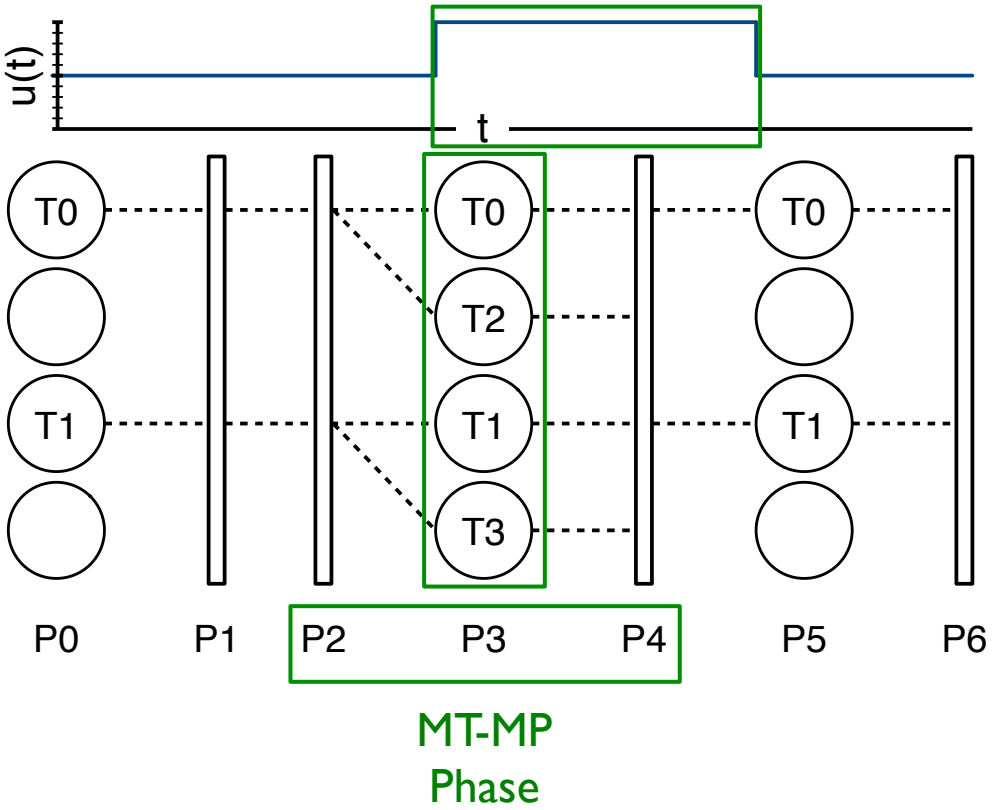


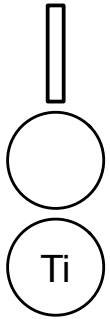
Parallel compute phase

Vacant processing element (PE)

PE occupied by task Ti

Today's Static Configurations: Under-Subscription



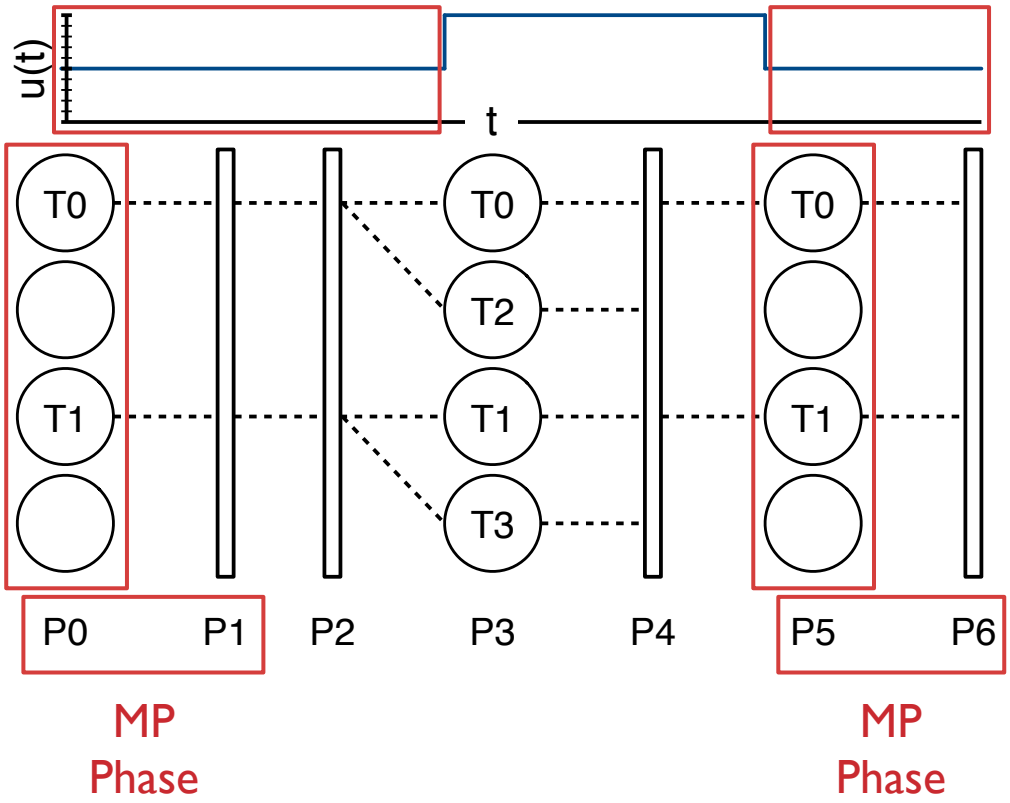


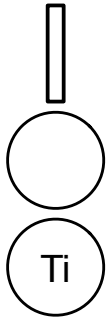
Parallel compute phase

Vacant processing element (PE)

PE occupied by task Ti

Today's Static Configurations: Under-Subscription



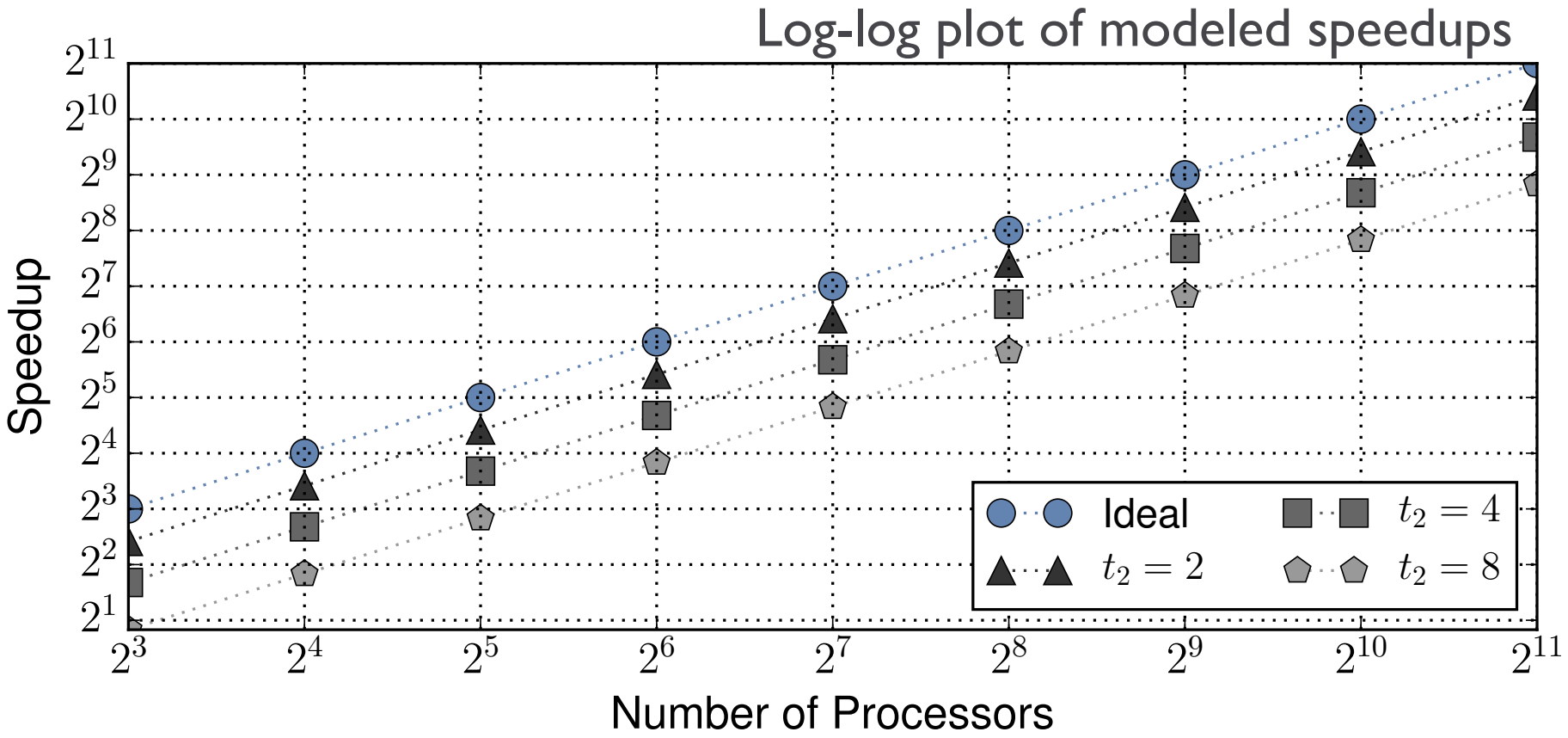


Parallel compute phase

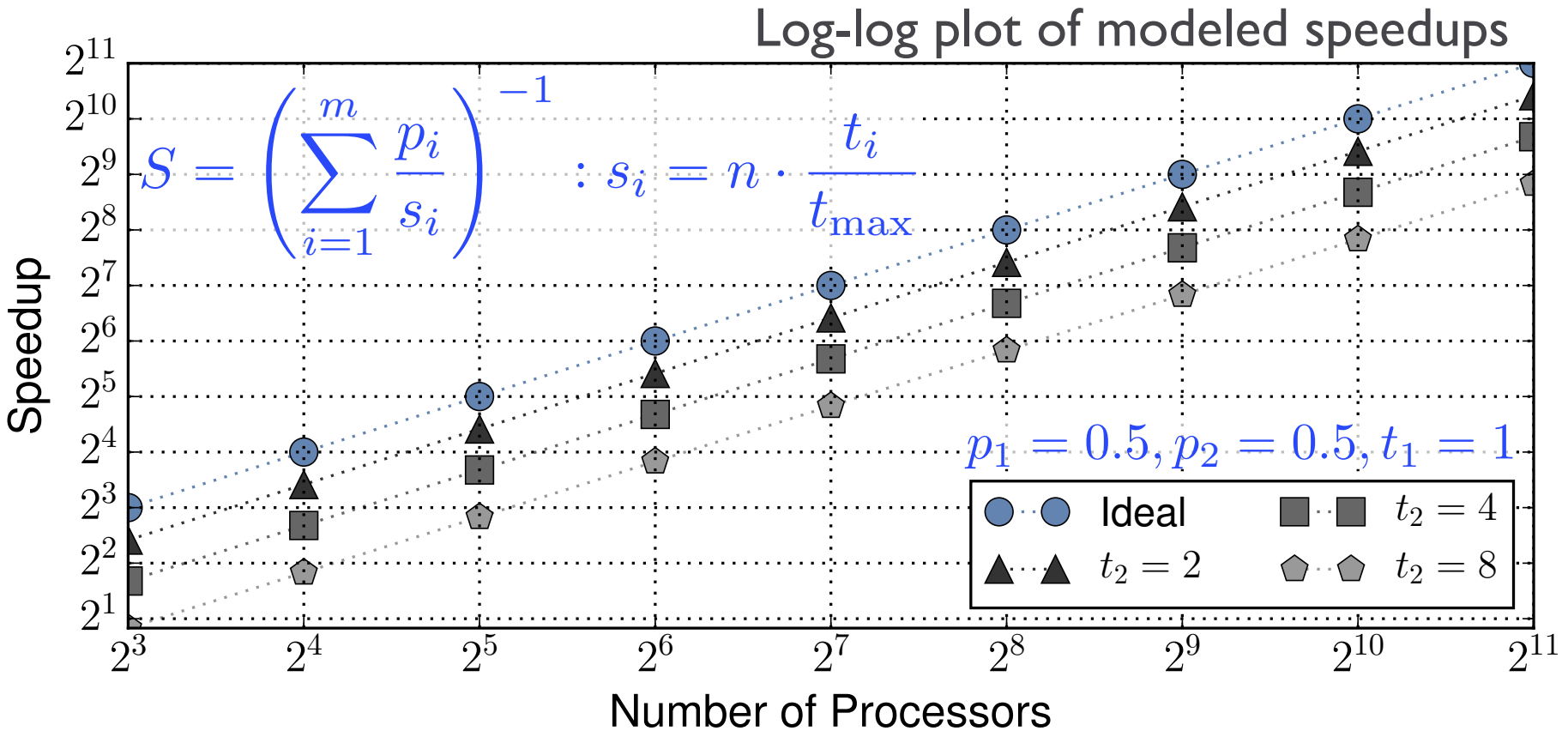
Vacant processing element (PE)

PE occupied by task Ti

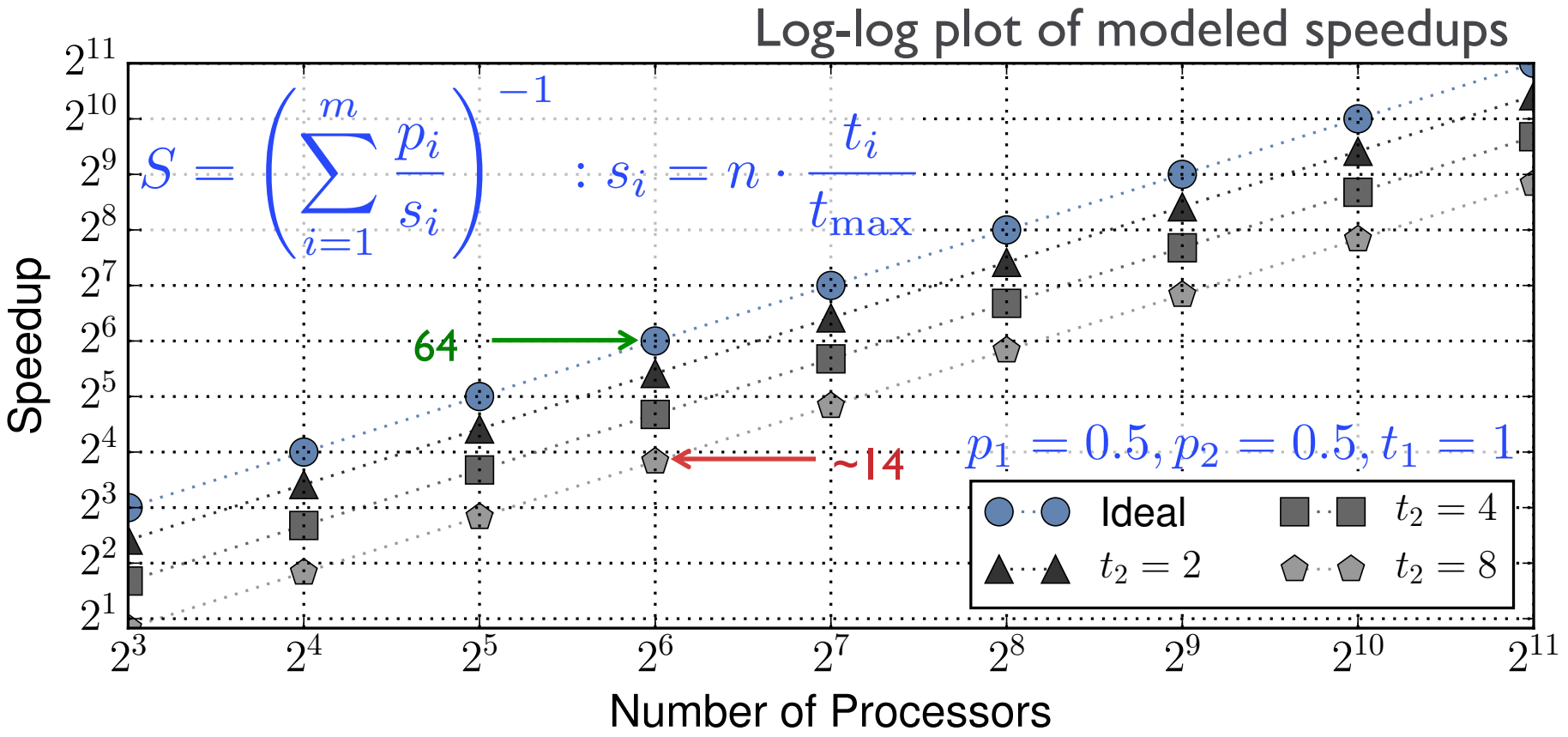
Lost Parallelism via Under-Subscription



Lost Parallelism via Under-Subscription



Lost Parallelism via Under-Subscription



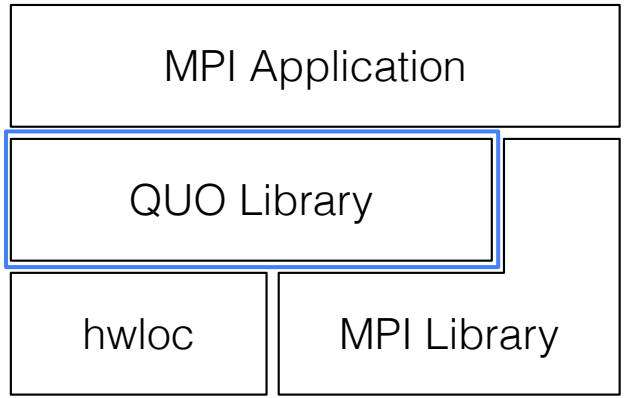
QUO Approach: Efficient Dynamic MPI+X

- Dynamic, run-time configuration conflict resolution for performant coupled thread-heterogeneous MP apps.

- Specifically, dynamic, coupled applications based on the Message Passing Interface (MPI)
  - MPI+X, where X is a Pthread-based runtime library
    - E.g., MPI-everywhere plus MPI+OpenMP or MPI+std::thread

QUO Features

- Programmer driven
- Hardware/software state queries (hardware topology, task config.)
- Efficient task reconfiguration (task placement, task affinity)
- Composable and general



- Open-source library (github.com/lanl/libquo)
- C, C++ (experimental), and Fortran interfaces

QUO Main Concepts

- Contexts (instance handles)

```
QUO_create(&ctx, MPI_COMM_WORLD)
```

```
QUO_free(ctx)
```

QUO Main Concepts

- Contexts (instance handles)
- Hardware and software environment queries

```
// Hardware topology queries
QUO_nobjs_by_type()
QUO_nobjs_in_type_by_type()
// Intra-task state queries
QUO_cpuset_in_type()
QUO_bound()
// Inter-task state queries
QUO_qids_in_type()
```

```
QUO_create(&ctx, MPI_COMM_WORLD)
// Dynamically determine target resource
tres = QUO_OBJ_NUMANODE
// Query hardware/software at run-time
QUO_auto_distrib(ctx, tres,
                 max_pe, &in_dset)
```

```
QUO_free(ctx)
```

QUO Main Concepts

- Contexts (instance handles)
- Hardware and software environment queries
- Dynamic process affinities
  - Stack-based semantics

```
QUO_create(&ctx, MPI_COMM_WORLD)
// Dynamically determine target resource
tres = QUO_OBJ_NUMANODE
// Query hardware/software at run-time
QUO_auto_distrib(ctx, tres,
                 max_pe, &in_dset)

    QUO_bind_push(ctx, tres)
    A_library_call(in_args, &result)
    QUO_bind_pop(ctx)

QUO_free(ctx)
```

QUO Main Concepts

- Contexts (instance handles)
- Hardware and software environment queries
- Dynamic process affinities
  - Stack-based semantics

- Efficient node-local process quiescence

```
QUO_create(&ctx, MPI_COMM_WORLD)
// Dynamically determine target resource
tres = QUO_OBJ_NUMANODE
// Query hardware/software at run-time
QUO_auto_distrib(ctx, tres,
                 max_pe, &in_dset)
```

```
    QUO_bind_push(ctx, tres)
    A_library_call(in_args, &result)
    QUO_bind_pop(ctx)

QUO_barrier(ctx)

QUO_free(ctx)
```

QUO Main Concepts

- Contexts (instance handles)
- Hardware and software environment queries
- Dynamic process affinities
  - Stack-based semantics
- Efficient node-local process quiescence
- Data dependencies

```
QUO_create(&ctx, MPI_COMM_WORLD)
// Dynamically determine target resource
tres = QUO_OBJ_NUMANODE
// Query hardware/software at run-time
QUO_auto_distrib(ctx, tres,
                 max_pe, &in_dset)
// Satisfy outstanding data dependencies
if (in_dset)
    QUO_bind_push(ctx, tres)
    A_library_call(in_args, &result)
    QUO_bind_pop(ctx)

QUO_barrier(ctx)
// Satisfy outstanding data dependencies
QUO_free(ctx)
```

QUO Main Concepts

- Contexts (instance handles)
- Hardware and software environment queries
- Dynamic process affinities
  - Stack-based semantics
- Efficient node-local process quiescence
- Data dependencies
- Policy management

```
QUO_create(&ctx, MPI_COMM_WORLD)
// Dynamically determine target resource
tres = QUO_OBJ_NUMANODE
// Query hardware/software at run-time
QUO_auto_distrib(ctx, tres,
                 max_pe, &in_dset)
// Satisfy outstanding data dependencies
if (in_dset)
    QUO_bind_push(ctx, tres)
    A_library_call(in_args, &result)
    QUO_bind_pop(ctx)

QUO_barrier(ctx)
// Satisfy outstanding data dependencies
QUO_free(ctx)
```

Practical Considerations

- Increased code complexity
  - Task quiescing + resumption → data remapping
- Encapsulating code regions
  - QUO_bind_push() + QUO_bind_pop()
- Broader applicability of affinity scheduling?

Questions

- Cost of run-time queries?
- Cost of task reconfiguration?

- Enough reclaimed parallelism to justify overheads?
  - Application runtime overhead
  - Development effort

Questions

- Cost of run-time queries?
- Cost of task reconfiguration?

Micro-Benchmarks

- Enough reclaimed parallelism to justify overheads?
  - Application runtime overhead
  - Development effort

Real Applications

Performance and Scaling Evaluation

- Micro-benchmark:
  - Quantify cost of key QUO operations

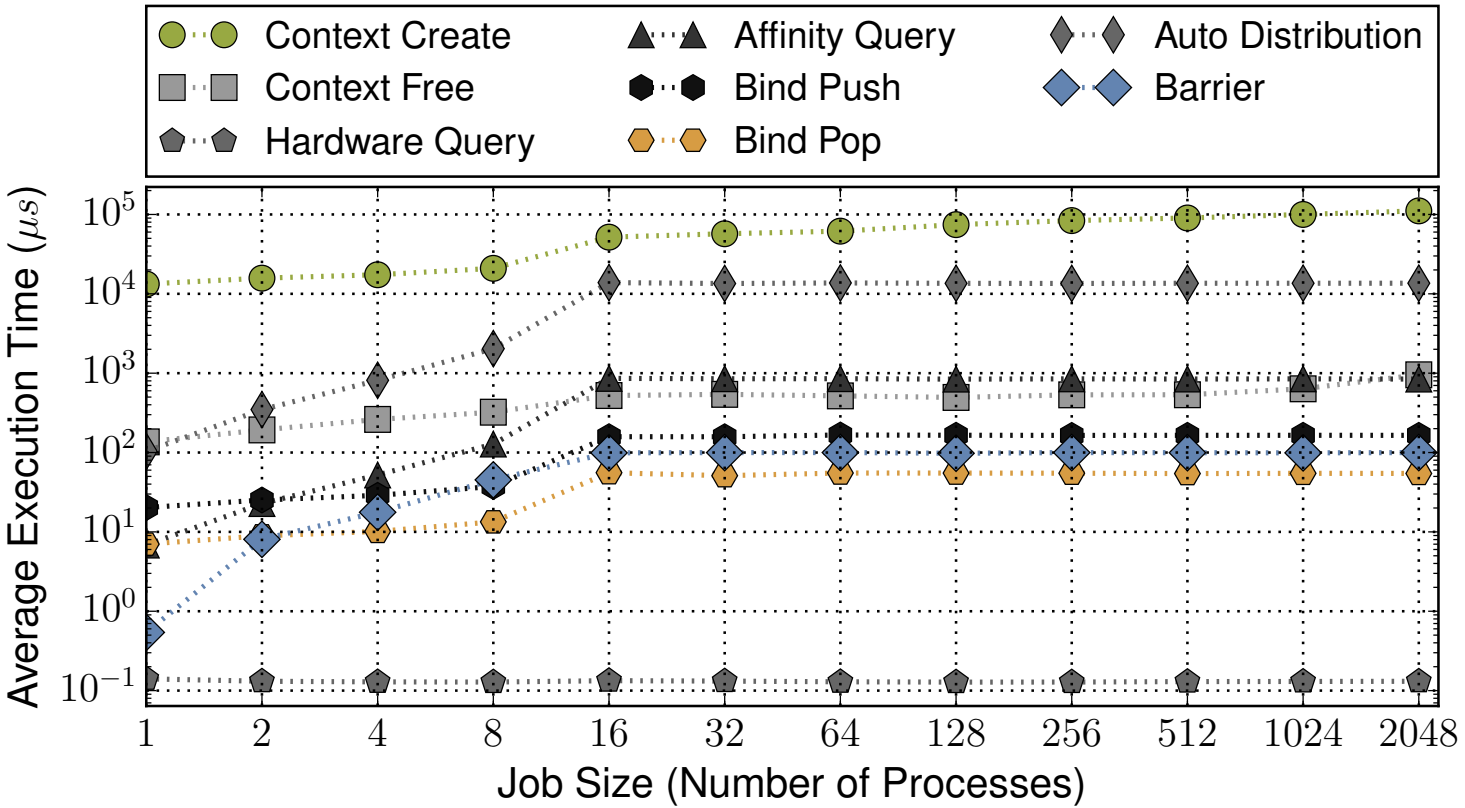
- Application integration
  - Three real-world production codes and inputs
  - Varied parallelization strategies, workloads, environments
    - 30 different configurations
  - Evaluated against predominant under-subscribed approach

Experimental Setup

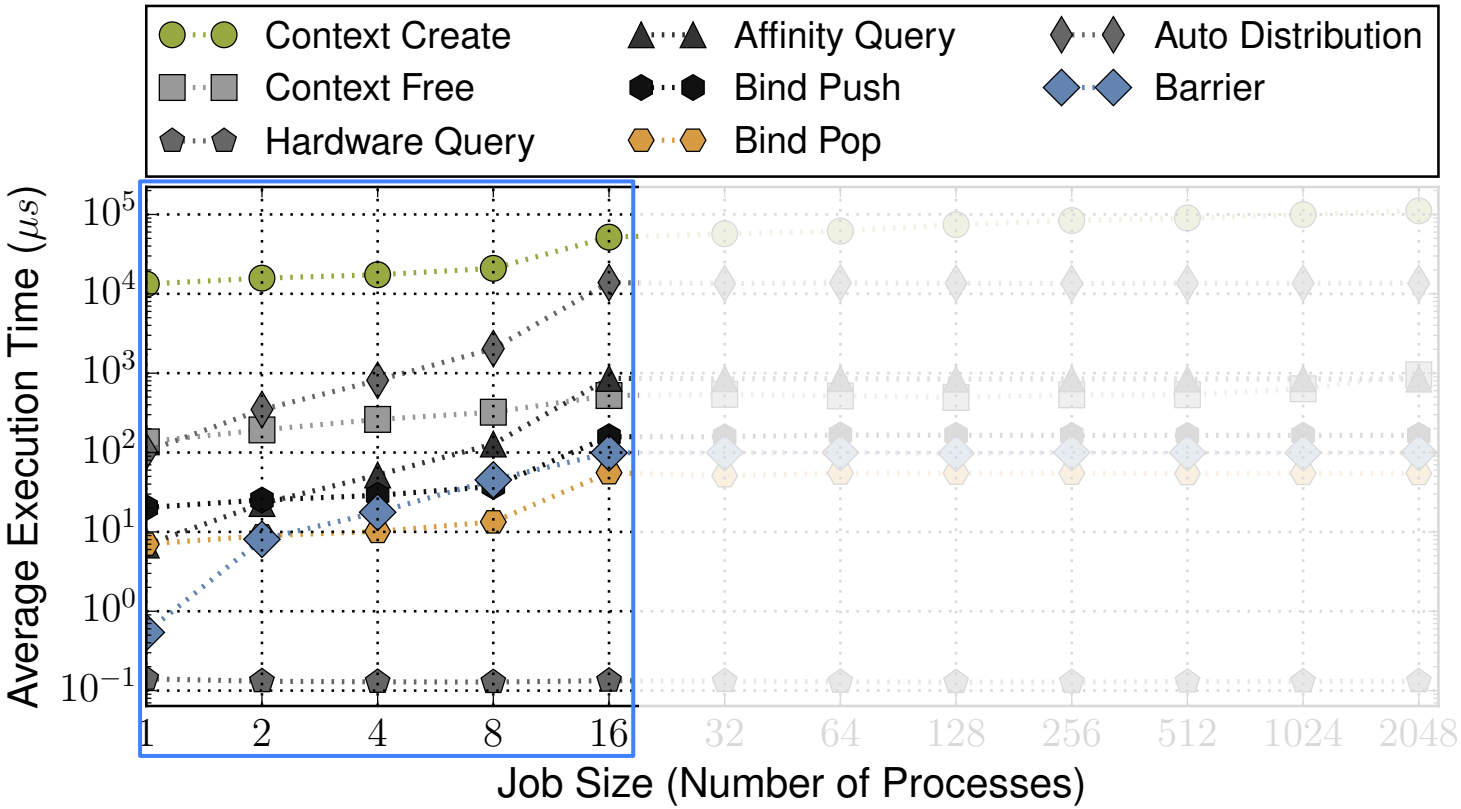
App. ID	CPU	MPI Library	Compiler	MT-MP Strategy	QUO Bindings
2MESH	AMD 6136	Cray-MPICH 7.0.1	Intel 15.0.4	MPI+OpenMP	Fortran
RAGE	Intel ES-2670	Open MPI 1.6.5	Intel 16.0.3	MPI+Kokkos	C
ESMD	Intel ES-2660	Open MPI 1.10.3	GCC 4.9.3	MPI+OpenMP	C++

QUO's Average Operational Latencies

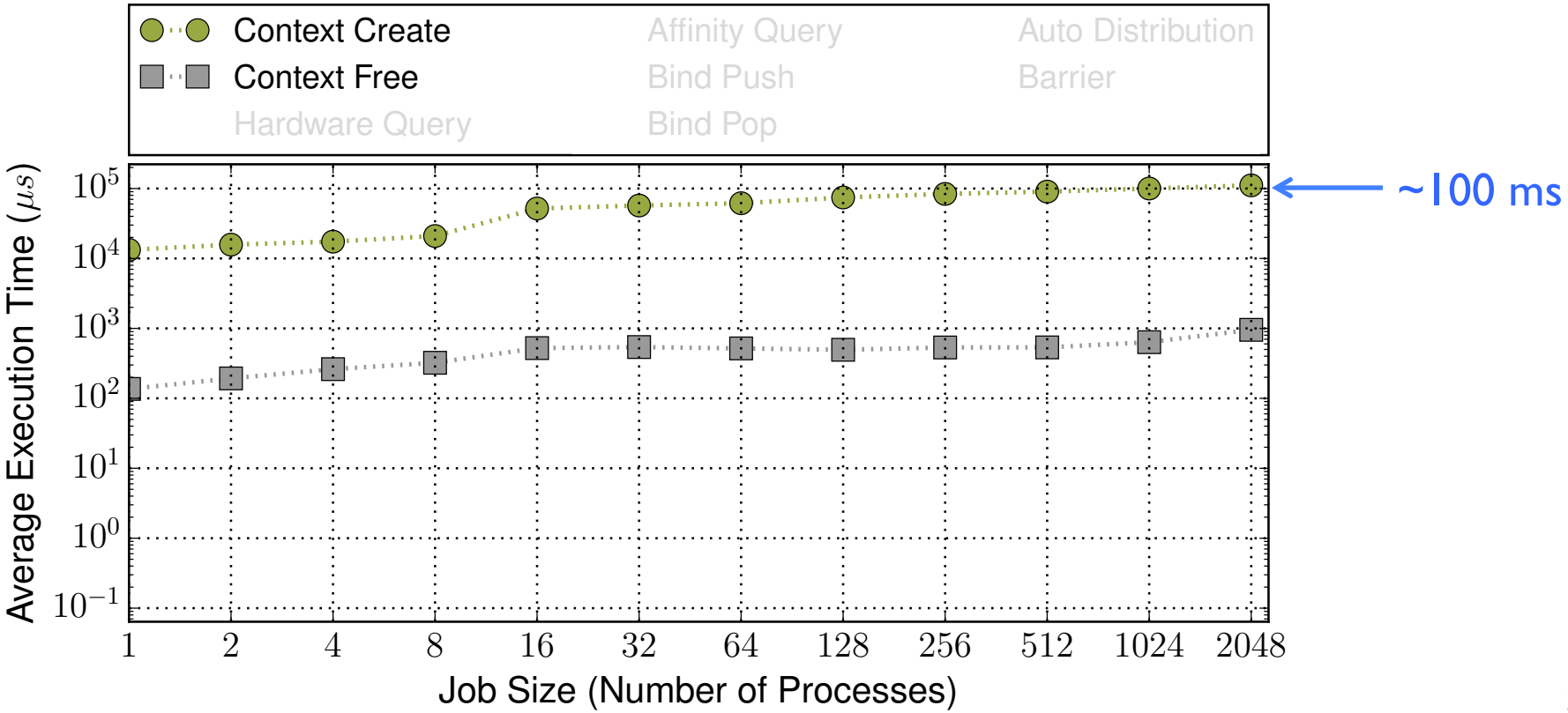
Results gathered from a Cray XE6 using Cray-MPICH 7.0.1



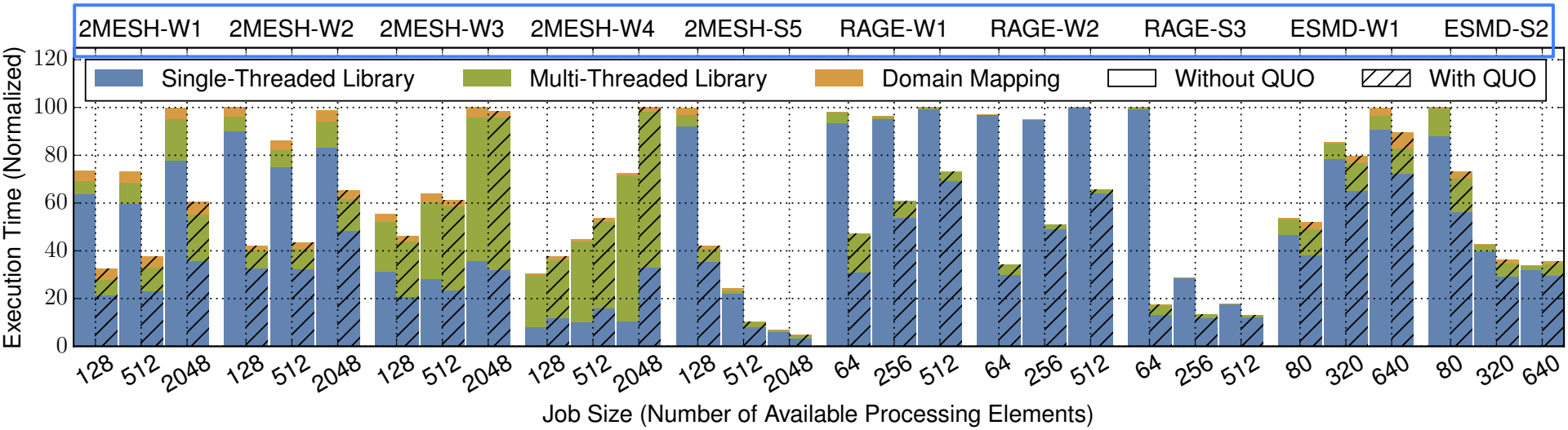
Node-Local Operational Latencies



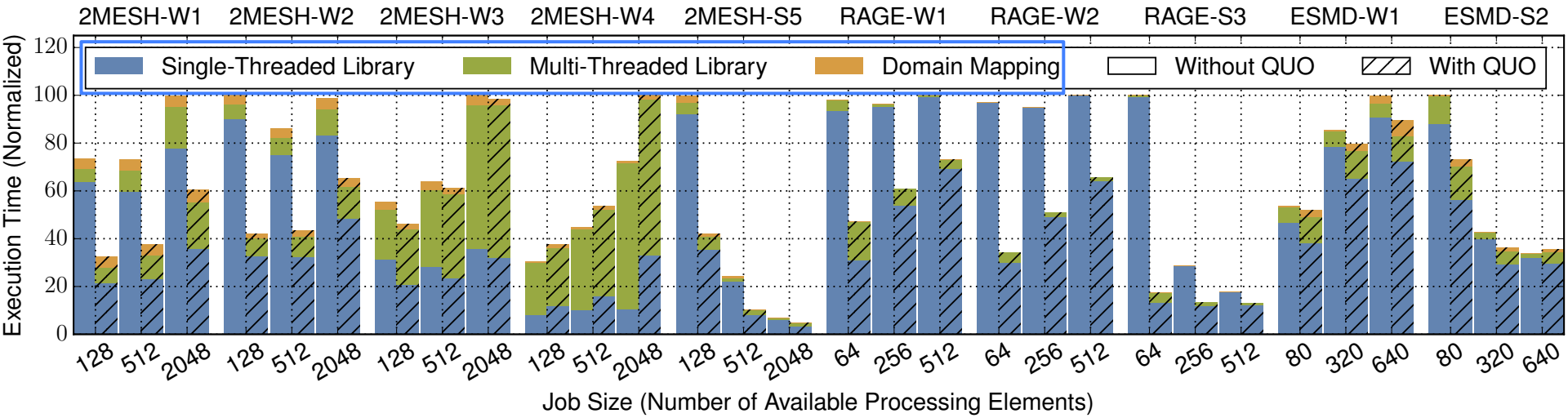
Global Operational Latencies



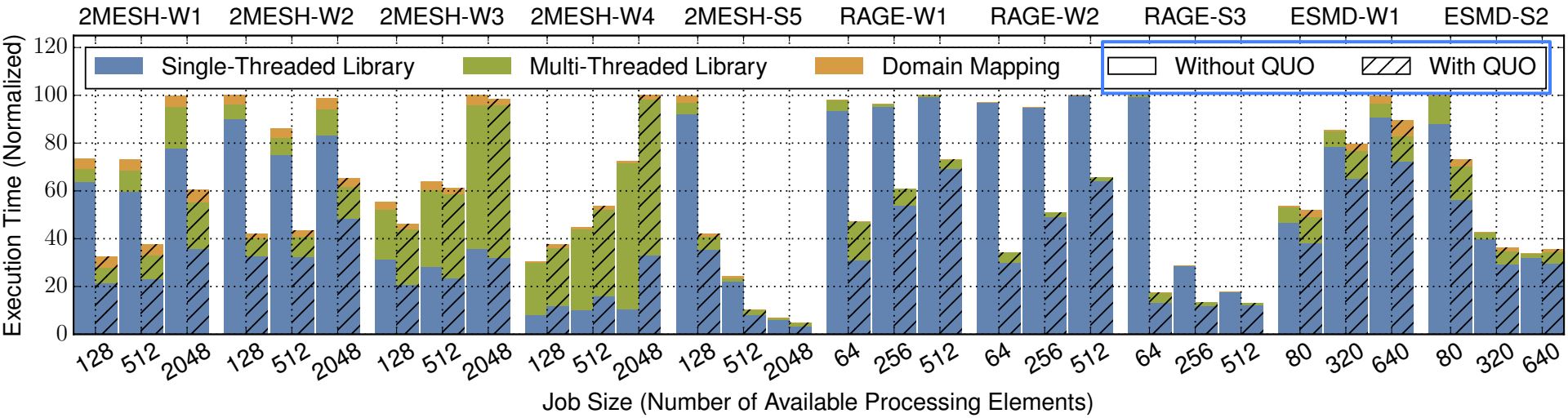
Average QUO-Based Application Performance



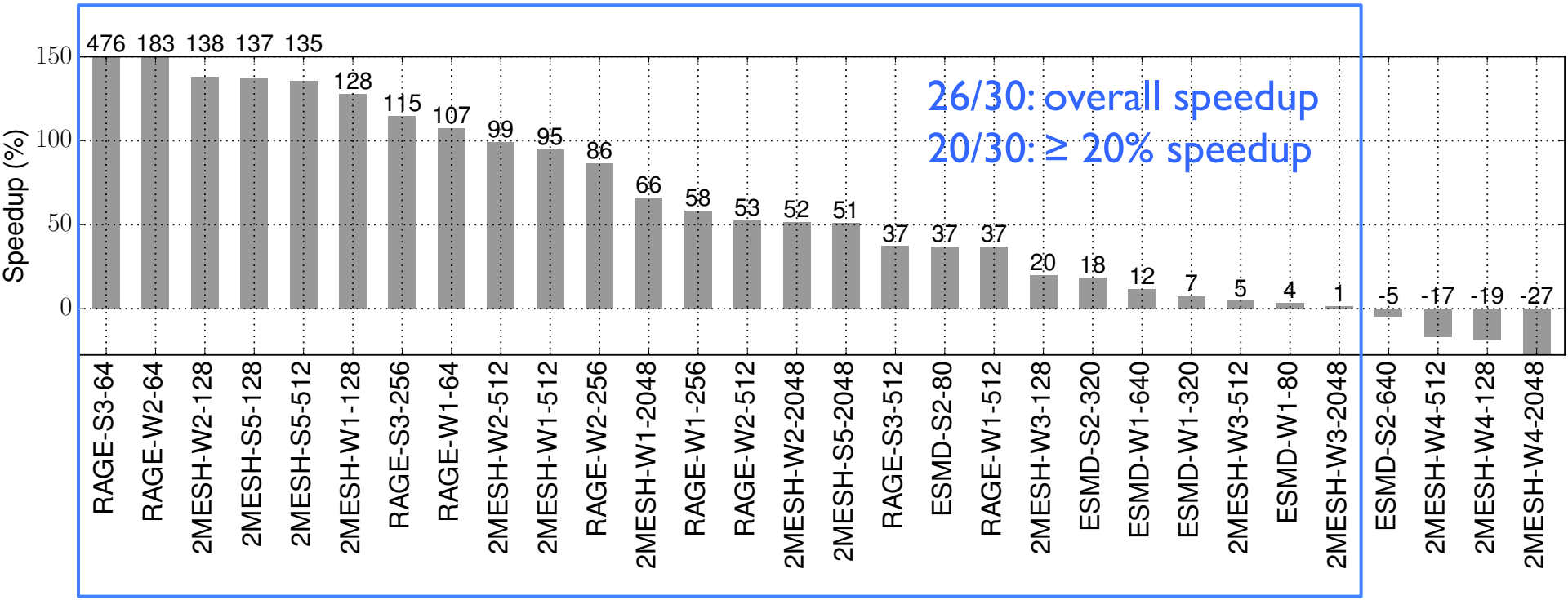
Average QUO-Based Application Performance



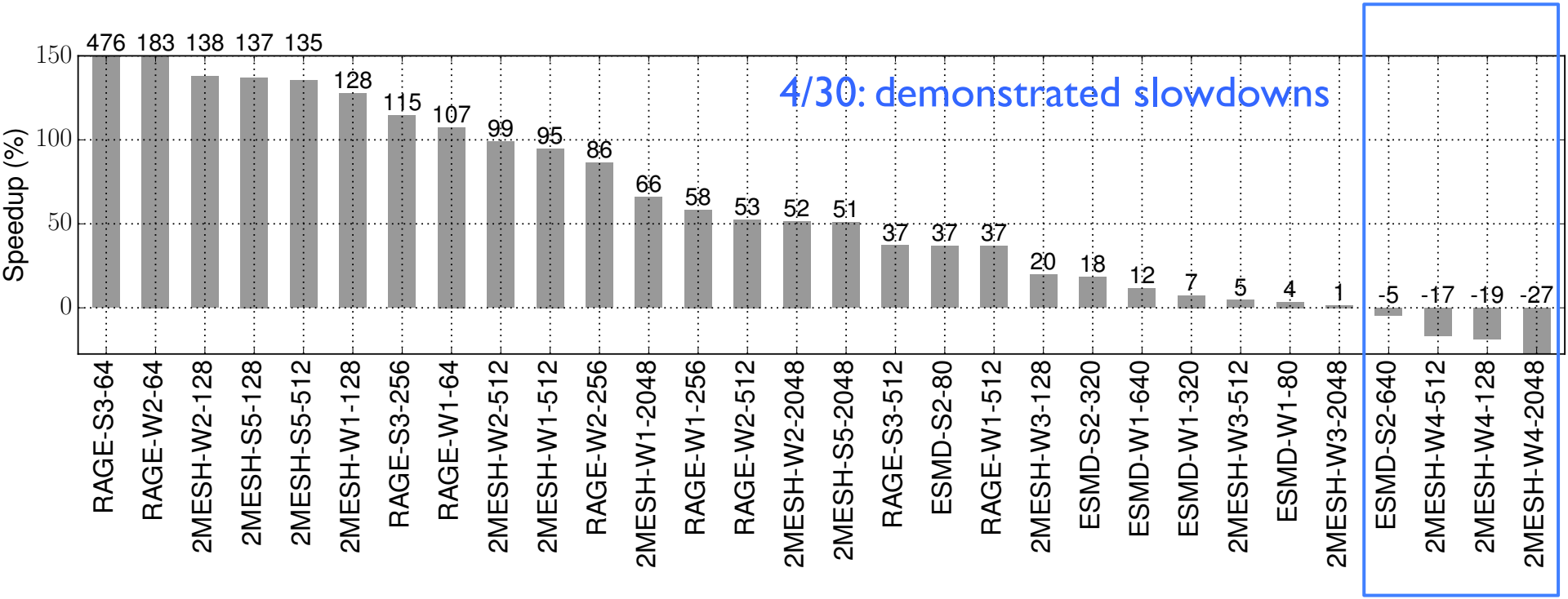
Average QUO-Based Application Performance



Summary of QUO-based Application Speedups



Summary of QUO-based Application Speedups



To QUO or not to QUO…

- Is my library in a thread-heterogeneous environment?
- Does my parallel library strong-scale well?
- Are its computational phases *long enough*?

Thank You

Questions?

This research was supported by the Advanced Simulation and Computing Program of the U.S. Department of Energy's NNSA and by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.