SAND2017-7549C

Tackling UQ in DARMA, a Programming Model for Task-Based Execution at Extreme-Scale

F. Rizzi , E. Phipps, D. Hollman, J. Lifflander, J. Wilke, A. Markosyan, H. Kolla, N. Slattengren, K. Teranishi, J. Stewart, R. Clay and J. Bennett

Sandia National Laboratories

QUIET17 - SISSA - Italy

Motivation

1 exaFlops: (10e18) calculations per second, supposedly arriving by 2023-2024

As of June 2017:

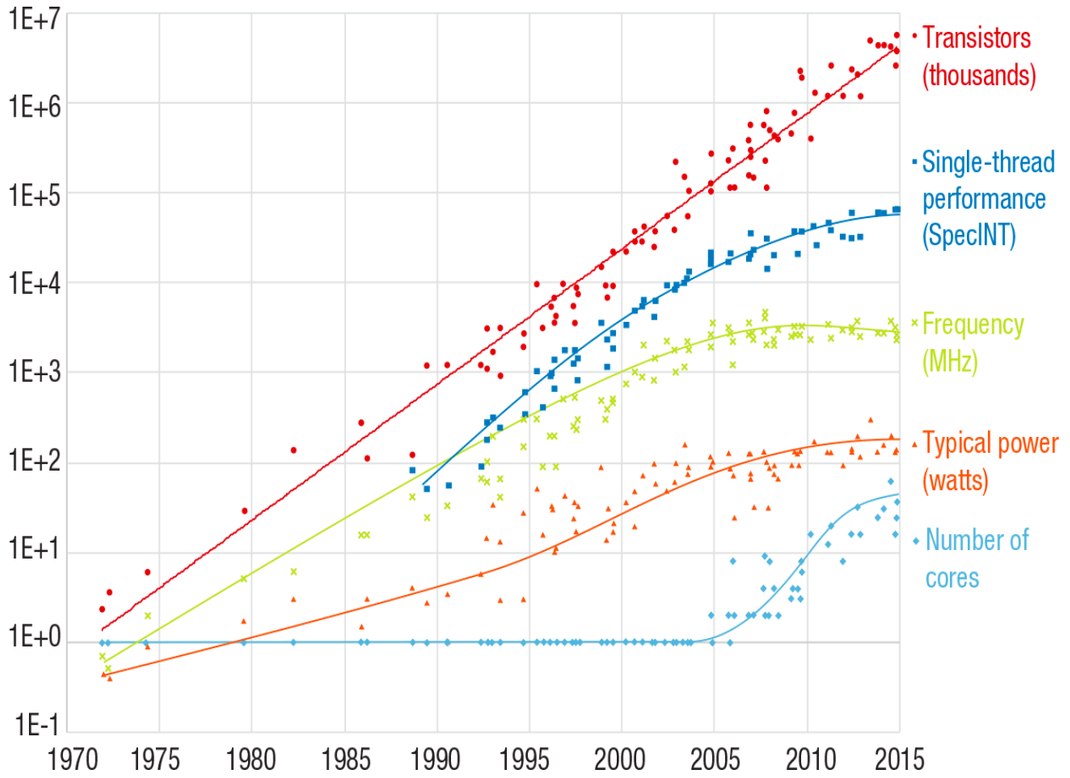
1. Sunway TaihuLight (China): 10,649,600 cores -- 125 PFlops -- 15.4 MW
2. Tianhe-2 (MilkyWay-2) (China): 3,120,000 cores -- 54 PFlops -- 17.8 MW
3. Piz Daint (Switzerland): 361,760 cores -- 25 PFlops -- 2.2 MW
4. Titan (USA): 560,640 cores -- 27 PFlops -- 8.2 MW

Challenges:

- Power consumption
- Complex (heterogeneous) architectures
- Unpredictable machines (resilience)
- Managing communication/computation
- Increasingly more dynamic workloads and machine performance

Can we ride the wave of current technology?

- Moore's observation: number of transistors on a chip doubles (nearly) every two years
- Dennard scaling: power density remains constant as transistors get smaller
- Dennard scaling broke down ~2005-2007
- Moore's trend is however alive and well
- Clock speeds are plateauing due to power and thermal limitations
- This is what has broken down: not the ability to etch smaller transistors, but the ability to drop the voltage and the current they need to operate reliably.



When you can't build outward any longer, build upward!

Clock frequency stalled, performance growth achieved by exponential growth in the number of processing elements per chip and growing hardware threading per core.

Increasing number of cores on the chip (expected to double every 18/24 months)

These trends motivate new programming abstractions that virtualize the notion of a core (implicit parallelism) and threading APIs with expanded semantics for thread control, placement, launching, and synchronization as well as scalable runtimes to manage massive numbers of threads.

Locality: Management of data locality is a first order concern.
Move computation to data, not viceversa.

Heterogeneity: accelerators, implicit data movement, heterogeneous machines.

Asynchrony: SPMD/bulk-synchronous programming models presume homogeneous performance across massively parallel systems. This will change.

Fault Tolerance: larger, more complex machines. Hundreds of millions of cores, circuits with feature sizes as small as 7 nm, and lower voltages than today.

DARMA

(Distributed Asynchronous Resilient Models and Applications)

C++ abstraction layer for asynchronous many-task (AMT) runtimes

Provides a set of abstractions to facilitate the expression of tasking that map to a variety of underlying AMT runtime system technologies.

Goals:

- Enables exploration of a variety of underlying runtime system technologies without changing application code.
- Facilitate the expression of coarse-grained tasking.

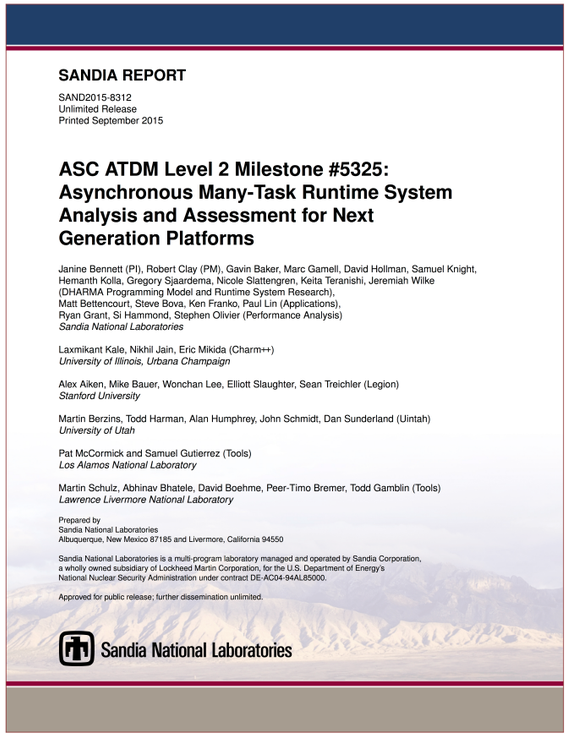
Applications are decomposed into small, transferable units of work (many tasks) with associated inputs (dependencies or data blocks) rather than simply decomposing at the process level (MPI ranks).

Is the coarse-grained, distributed memory analog of instruction-level parallelism, extending the concepts of data pre-fetching, out-of-order task execution based on dependency analysis, and even branch prediction (speculative execution). Rather than executing in a well-defined order, tasks execute when inputs become available.

An AMT model aims to leverage all available task and pipeline parallelism, rather just relying on basic data parallelism for concurrency.

Enables the overlap of communication and computation as well as asynchronous load balancing strategis

- Broad survey of many AMT runtime systems
- Deep dive on Charm++, Legion, Uintah
- Programmability:
  Does this runtime enable efficient expression of workloads?
- Performance:
  How performant is this runtime for our workloads on current platforms and how well suited is this runtime to address future architecture challenges?
- Mutability:
  What is the ease of adopting this runtime and modifying it to suit our code needs?



SANDIA REPORT

SAND2015-8312
Unlimited Release
Printed September 2015

ASC ATDM Level 2 Milestone #5325: Asynchronous Many-Task Runtime System Analysis and Assessment for Next Generation Platforms

Janine Bennett (PI), Robert Clay (PM), Gavin Baker, Marc Gamell, David Hollman, Samuel Knight, Hemanth Kolla, Gregory Sjaardema, Nicole Slattengren, Keita Teranishi, Jeremiah Wilke
(DHARMA Programming Model and Runtime System Research),
Matt Bettencourt, Steve Bova, Ken Franko, Paul Lin (Applications),
Ryan Grant, SI Hammond, Stephen Olivier (Performance Analysis)
Sandia National Laboratories

Laxmikant Kale, Nikhil Jain, Eric Mikida (Charm++)
University of Illinois, Urbana Champaign

Alex Aiken, Mike Bauer, Wonchan Lee, Elliott Slaughter, Sean Treichler (Legion)
Stanford University

Martin Berzins, Todd Harman, Alan Humphrey, John Schmidt, Dan Sunderland (Uintah)
University of Utah

Pat McCormick and Samuel Gutierrez (Tools)
Los Alamos National Laboratory

Martin Schulz, Abhinav Bhatele, David Boehme, Peer-Timo Bremer, Todd Gamblin (Tools)
Lawrence Livermore National Laboratory

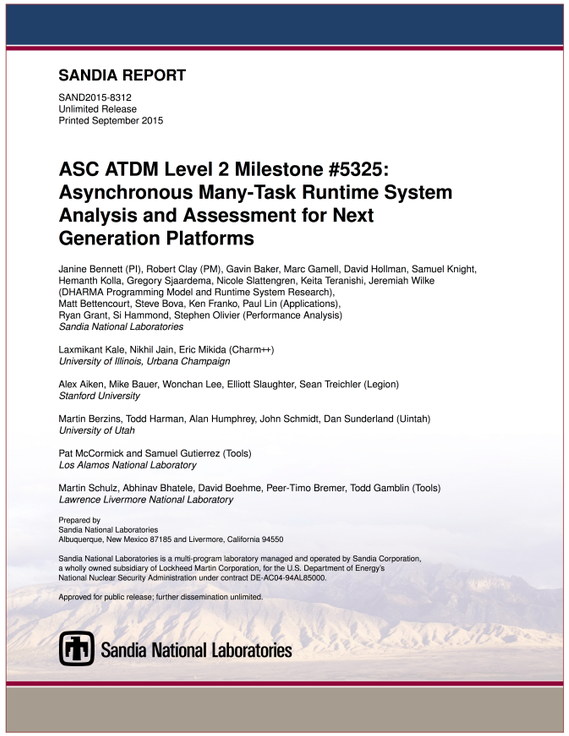
Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Approved for public release; further dissemination unlimited.

Sandia National Laboratories

- AMT systems show great promise
- No common user-level APIs
- Need for best practices and standards
- Survey recommendations led to DARMA
- C++ abstraction layer for AMT runtimes
- Requirements driven by Sandia ATDM applications
- A single user-level API
- Support multiple AMT runtimes to begin identification of best practices



SANDIA REPORT

SAND2015-8312
Unlimited Release
Printed September 2015

ASC ATDM Level 2 Milestone #5325: Asynchronous Many-Task Runtime System Analysis and Assessment for Next Generation Platforms

Janine Bennett (PI), Robert Clay (PM), Gavin Baker, Marc Gamell, David Hollman, Samuel Knight, Hemanth Kolla, Gregory Sjaardema, Nicole Slattengren, Keita Teranishi, Jeremiah Wilke
(DHARMA Programming Model and Runtime System Research),
Matt Bettencourt, Steve Bova, Ken Franko, Paul Lin (Applications),
Ryan Grant, SI Hammond, Stephen Olivier (Performance Analysis)
Sandia National Laboratories

Laxmikant Kale, Nikhil Jain, Eric Mikida (Charm++)
University of Illinois, Urbana Champaign

Alex Aiken, Mike Bauer, Wonchan Lee, Elliott Slaughter, Sean Treichler (Legion)
Stanford University

Martin Berzins, Todd Harman, Alan Humphrey, John Schmidt, Dan Sunderland (Uintah)
University of Utah

Pat McCormick and Samuel Gutierrez (Tools)
Los Alamos National Laboratory

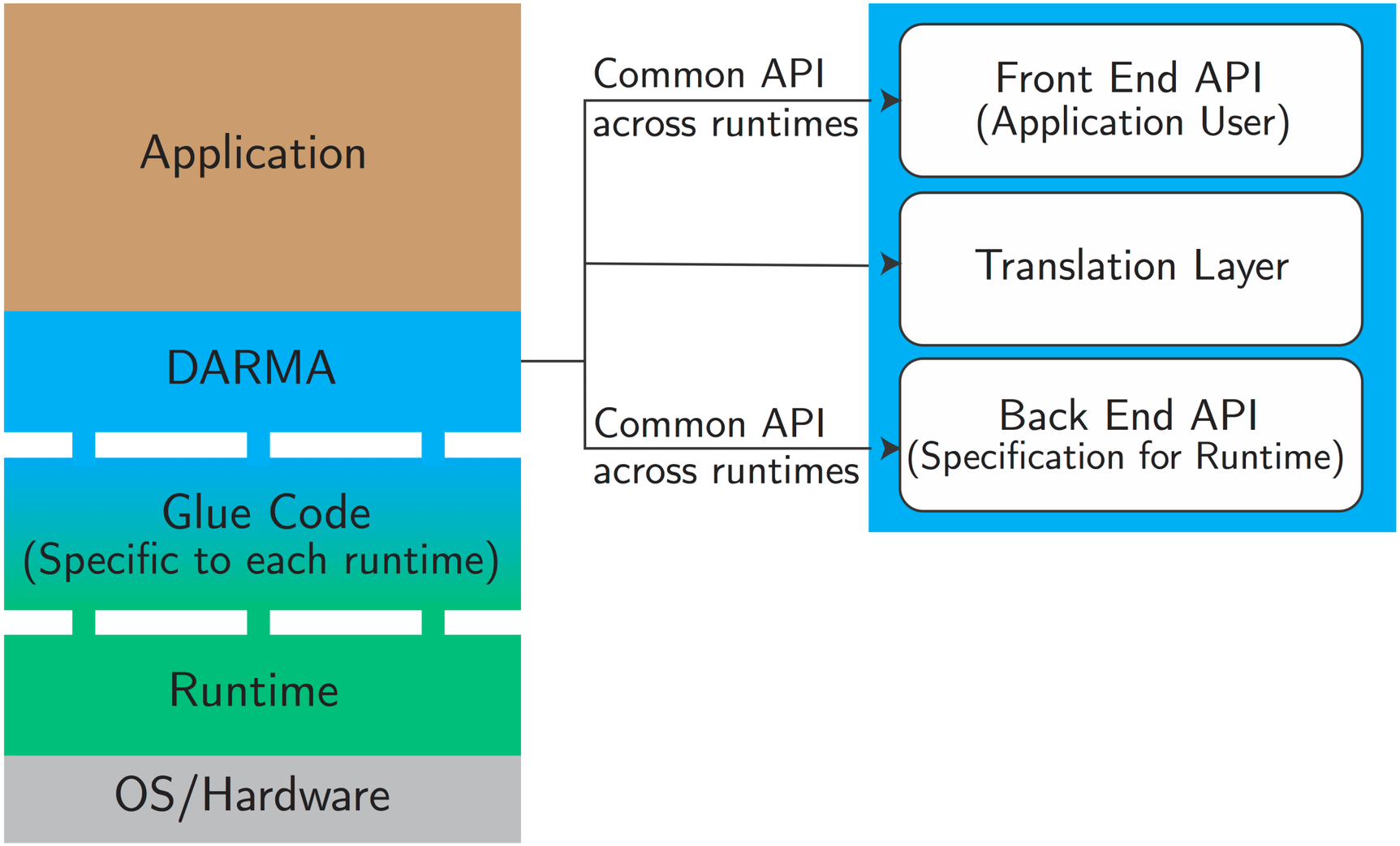
Martin Schulz, Abhinav Bhatele, David Boehme, Peer-Timo Bremer, Todd Gamblin (Tools)
Lawrence Livermore National Laboratory

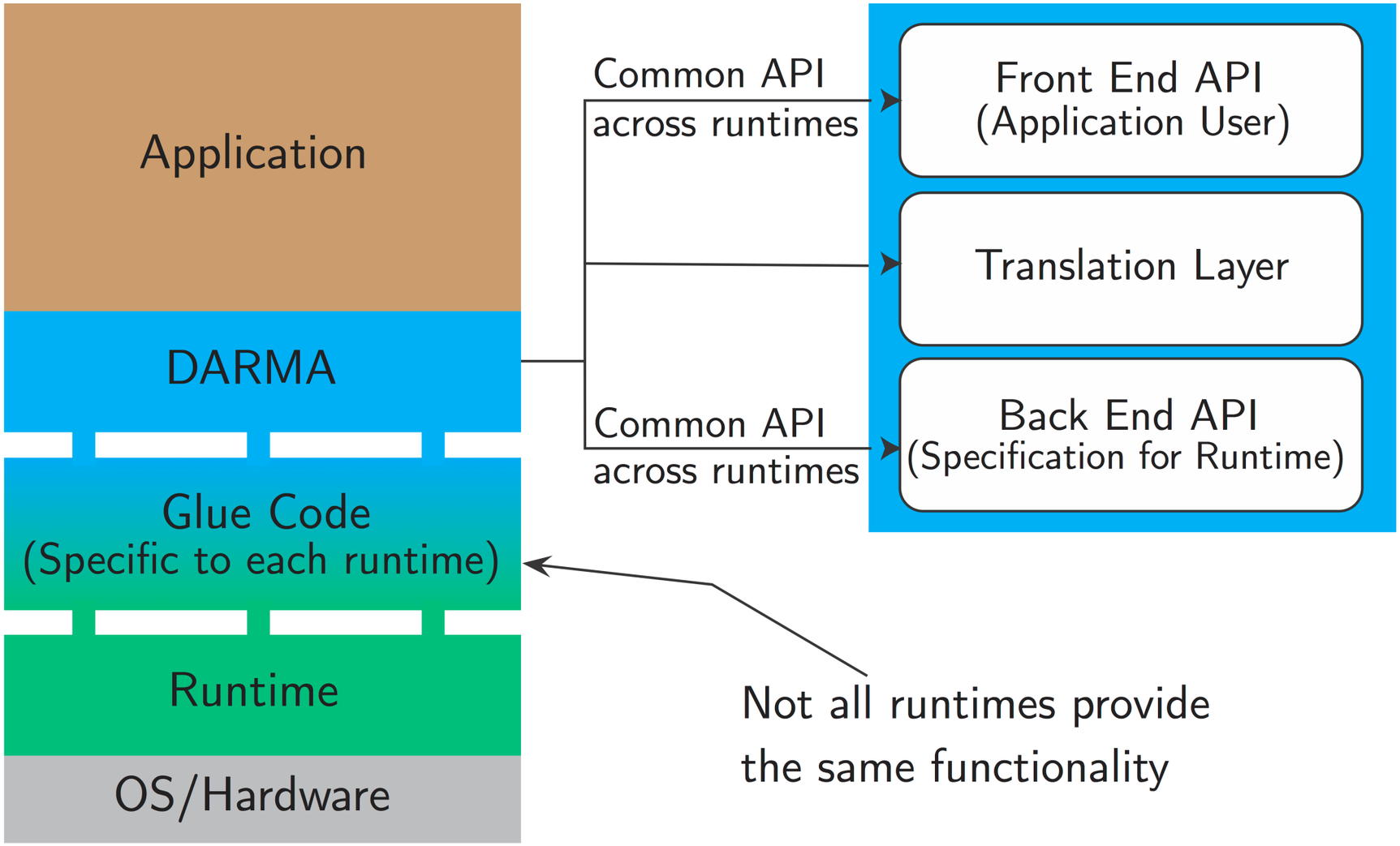
Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

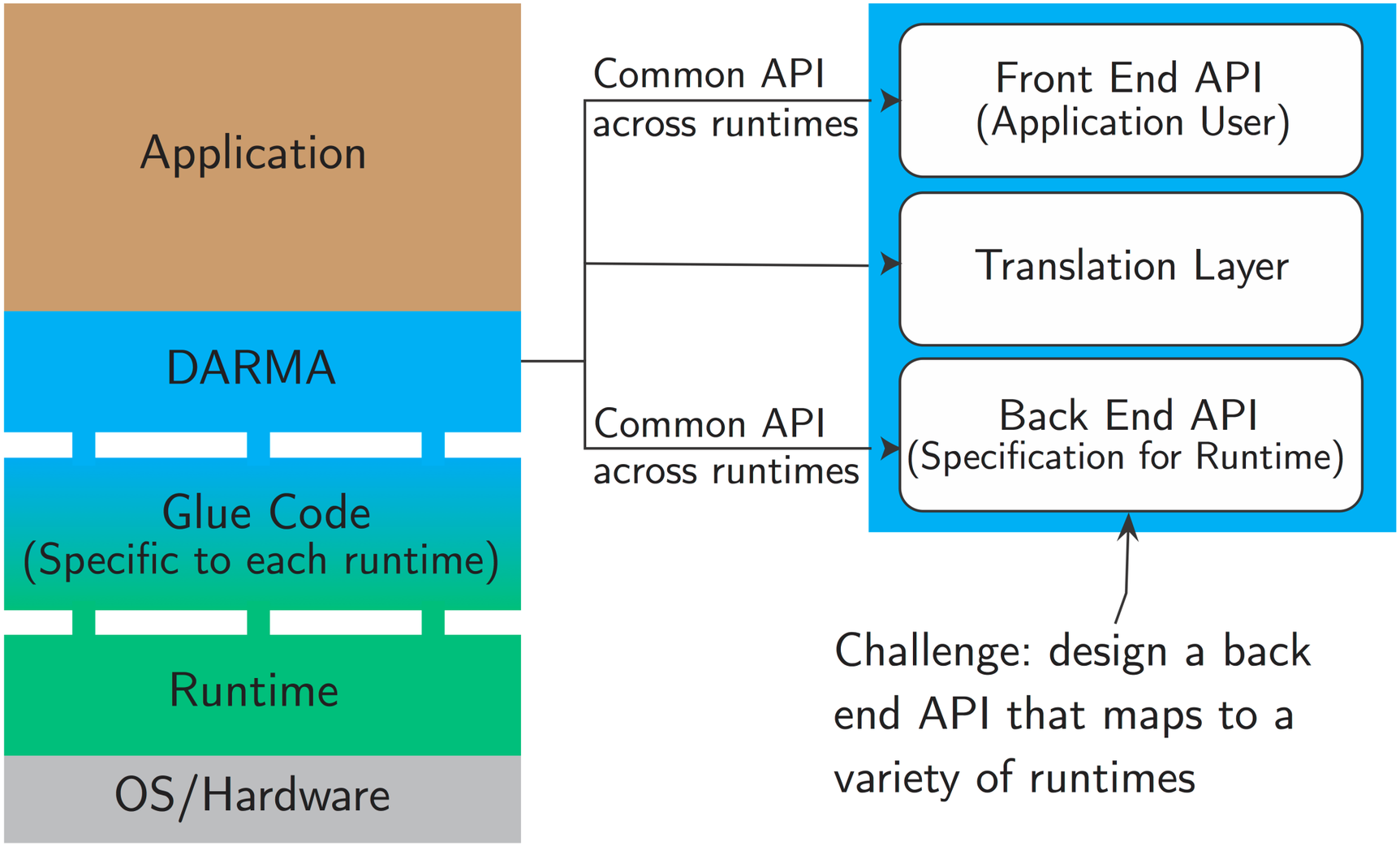
Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Approved for public release; further dissemination unlimited.

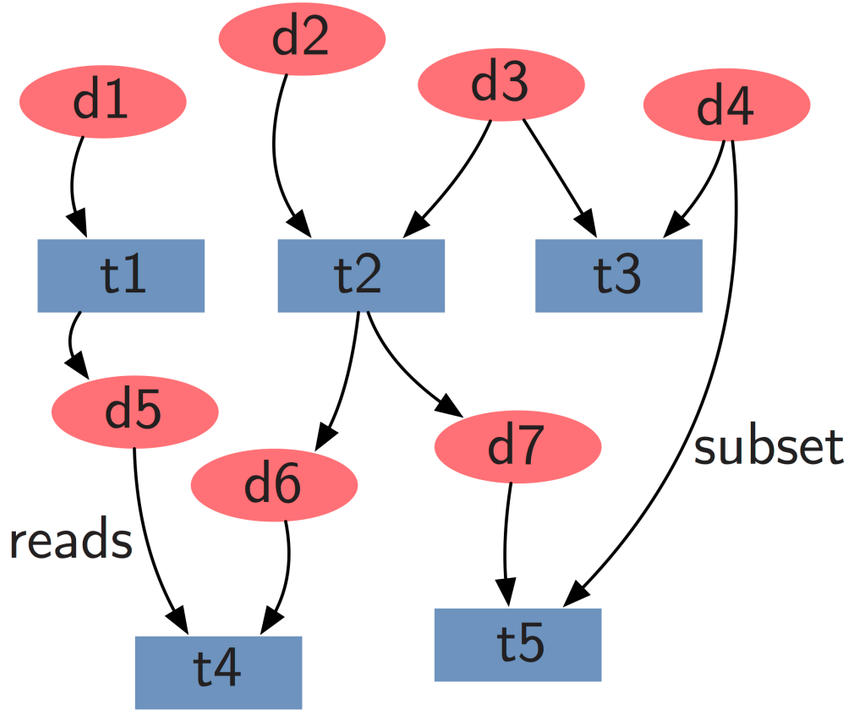
Sandia National Laboratories


Application
DARMA
Glue Code
(Specific to each runtime)
Runtime
OS/Hardware
Common API
across runtimes
Common API
across runtimes
Front End API
(Application User)
Translation Layer
Back End API
(Specification for Runtime)


Application
DARMA
Glue Code
(Specific to each runtime)
Runtime
OS/Hardware
Common API
across runtimes
Common API
across runtimes
Front End API
(Application User)
Translation Layer
Back End API
(Specification for Runtime)
Not all runtimes provide
the same functionality


Application
DARMA
Glue Code
(Specific to each runtime)
Runtime
OS/Hardware
Common API
across runtimes
Common API
across runtimes
Front End API
(Application User)
Translation Layer
Back End API
(Specification for Runtime)
Challenge: design a back end API that maps to a variety of runtimes

- AMT runtimes operate with a directed acyclic graph (DAG)
- Captures relationships between application data and inter-dependent tasks
- DAGs can be annotated to capture additional information
  - Tasks' read/write usage of data
  - Task needs a subset of data
- Additional information enables runtime to reason more completely about
  - When and where to execute a task
  - Whether to load balance
- Existing runtimes leverage DAGs with varying degrees of annotation



Serial code

```
void get_foo(int& val) {
  /* some work... */
  val = 42; }
void get_bar(int& val) {
  /*...*/
  val = 73; }
void print(int a, int b) {
  cout << a << ", "
       << b << endl;
}
int main() {
  int foo, bar;
  get_foo(foo);
  get_bar(bar);
  print(foo, bar);
}
```

Output: 42, 73

Explicit threads

```
static int foo, bar;
void get_foo() {
  /* some work... */
  foo = 42; }
void get_bar() {
  /*...*/
  bar = 73; }
void print() {
  cout << foo << ", "
       << bar << endl;
}
int main() {
  auto thr_foo = std::thread(get_foo);
  auto thr_bar = std::thread(get_bar);
  thr_foo.join();
  thr_bar.join();
  print();
}
```

Output: 42, 73

Using `async-future`:

```
int get_foo() {
  /* some work... */
  return 42; }
int get_bar() {
  /* ... */
  return 73; }
void print(future a, future b) {
  cout << a.get() << ", "
       << b.get() << endl;
}
int main() {
  auto foo = std::async(get_foo);
  auto bar = std::async(get_bar);
  auto done = std::async(print,
              move(foo), move(bar));
  done.wait();
}
```

Output: 42, 73

- Direct extraction of concurrency based on the sequence of data usage
- Conservative because it is "safe by default"
- Enabling runtime-based approaches rather than auto-magic compilers
- There is existing related research (e.g., Legion, OpenMP 4.5)

The function signature itself (from the sequential implementation) can serve as a concurrency specification!

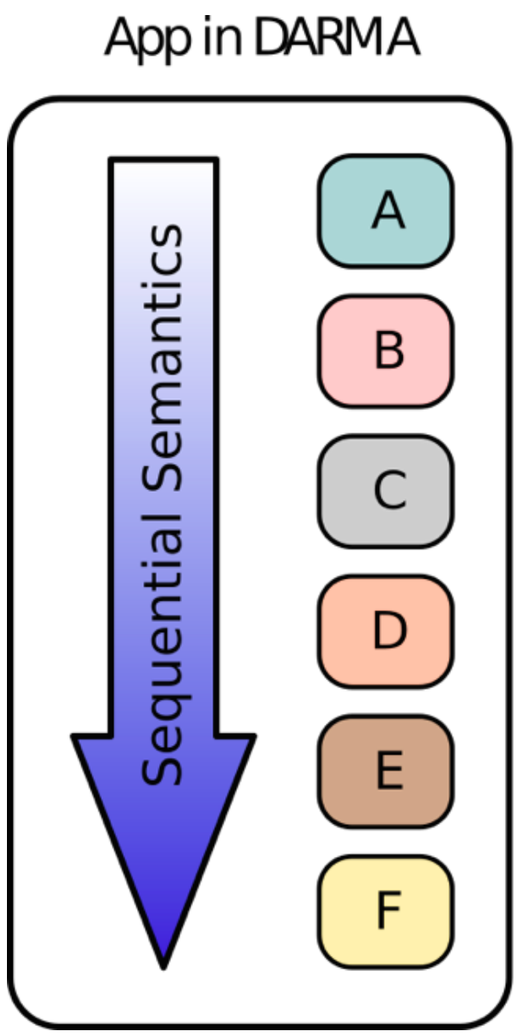
Serial code

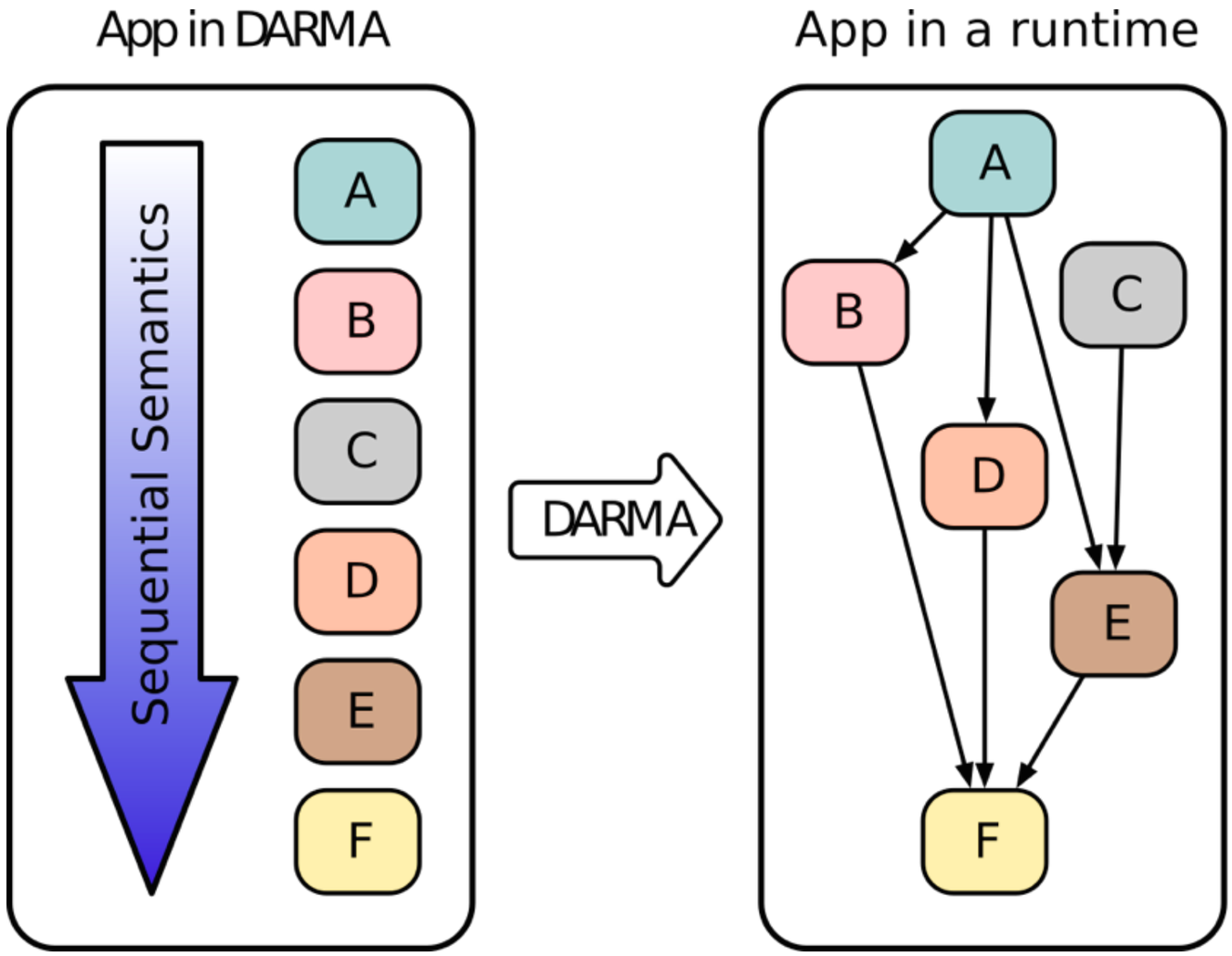
```
void get_foo(int& val) { /*...*/ val = 42; }
void get_bar(int& val) { /*...*/ val = 73; }
void print(int a, int b) {
  cout << a << ", " << b << endl;
}

int main() {
  int foo, bar;
  get_foo(foo);
  get_bar(bar);
  print(foo, bar);
}
```

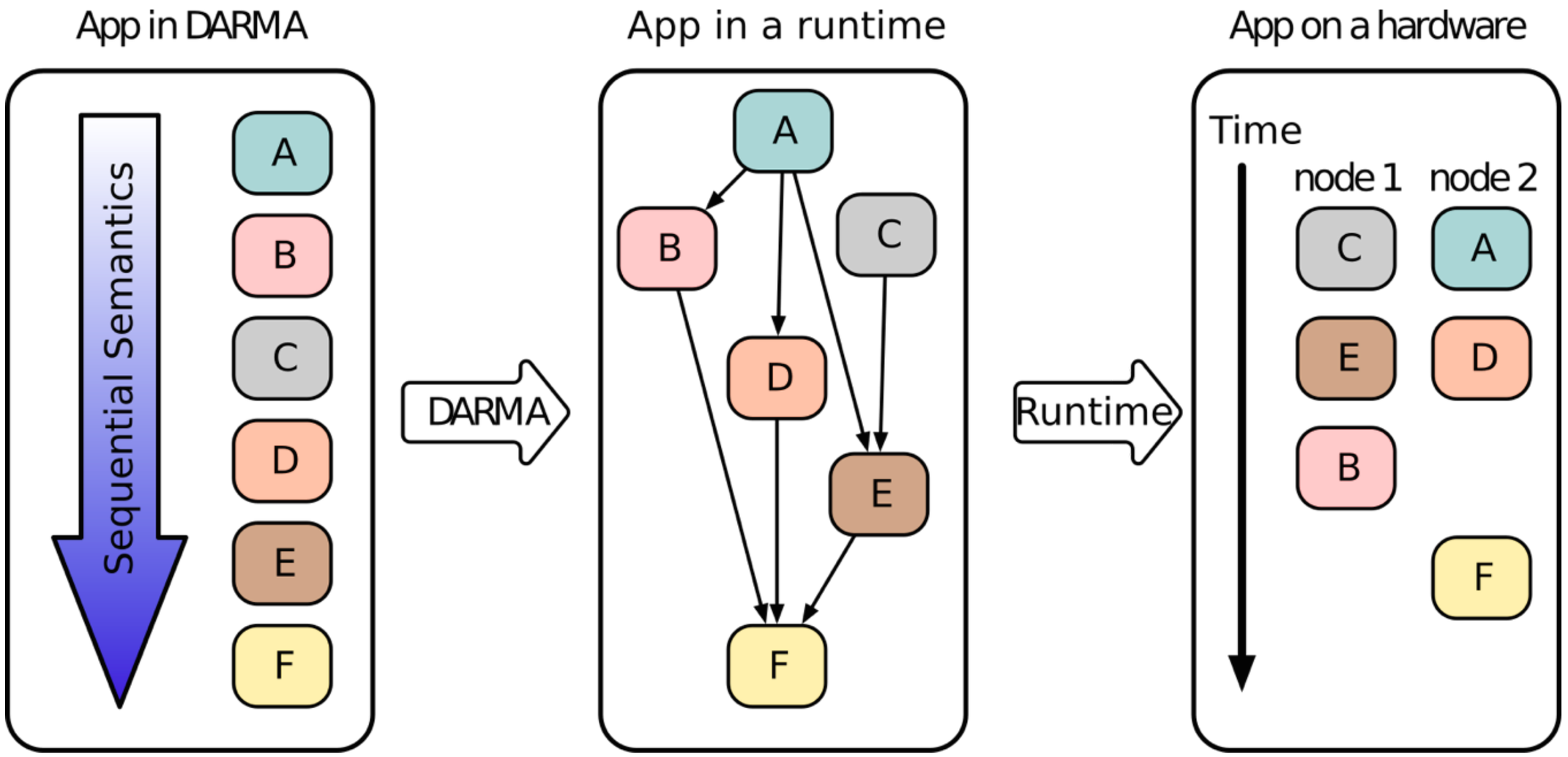
(quasi) DARMA code

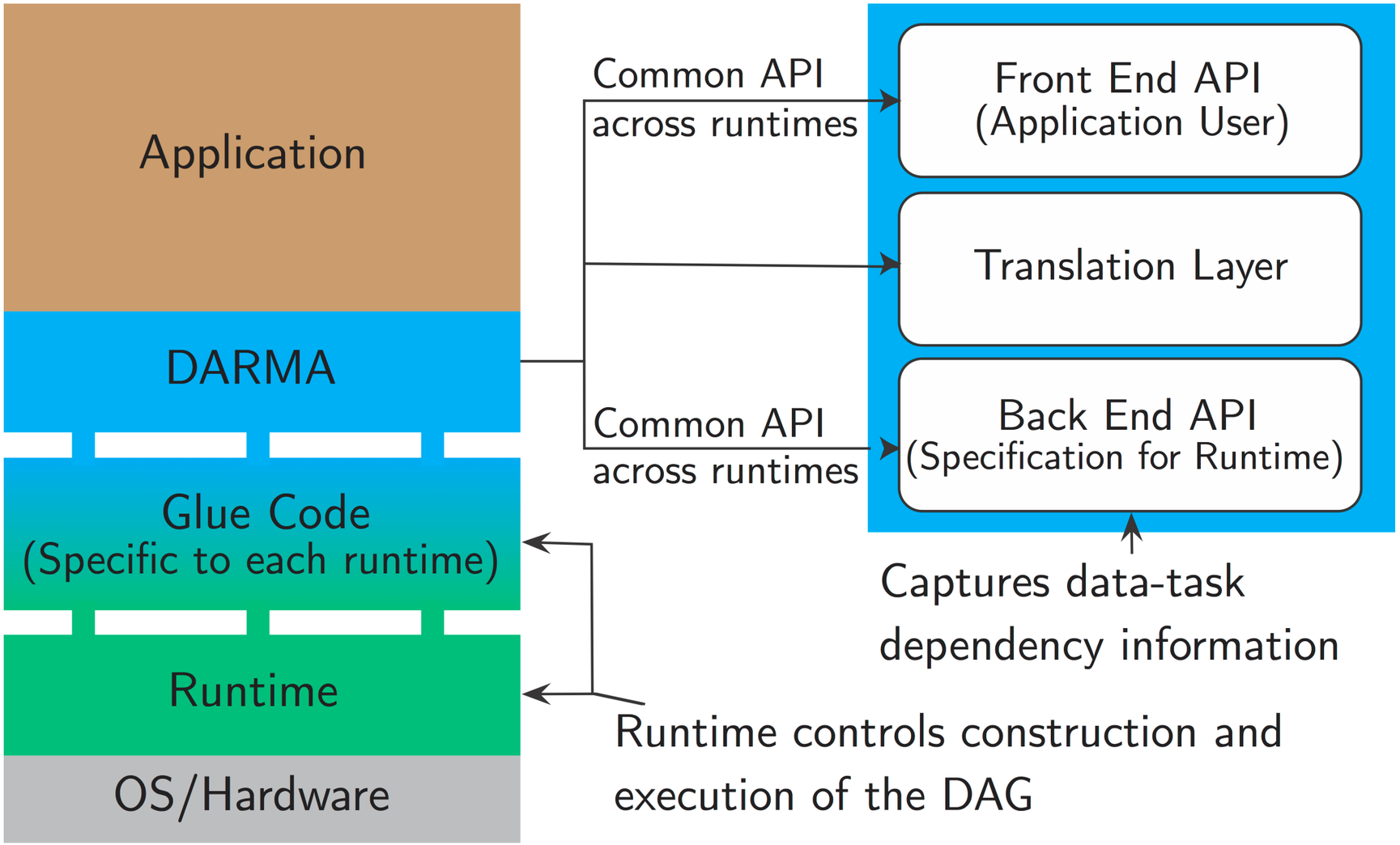
```
void get_foo(int& val) { /*... */ val = 42; }
void get_bar(int& val) { /*...*/ val = 73; }
void print(int a, int b) {
  cout << a << ", " << b << endl;
}

int main() {
  async_ptr<int> foo, bar;
  mpass::async(get_foo, foo);
  mpass::async(get_bar, bar);
  mpass::async(print, foo, bar);
}
```

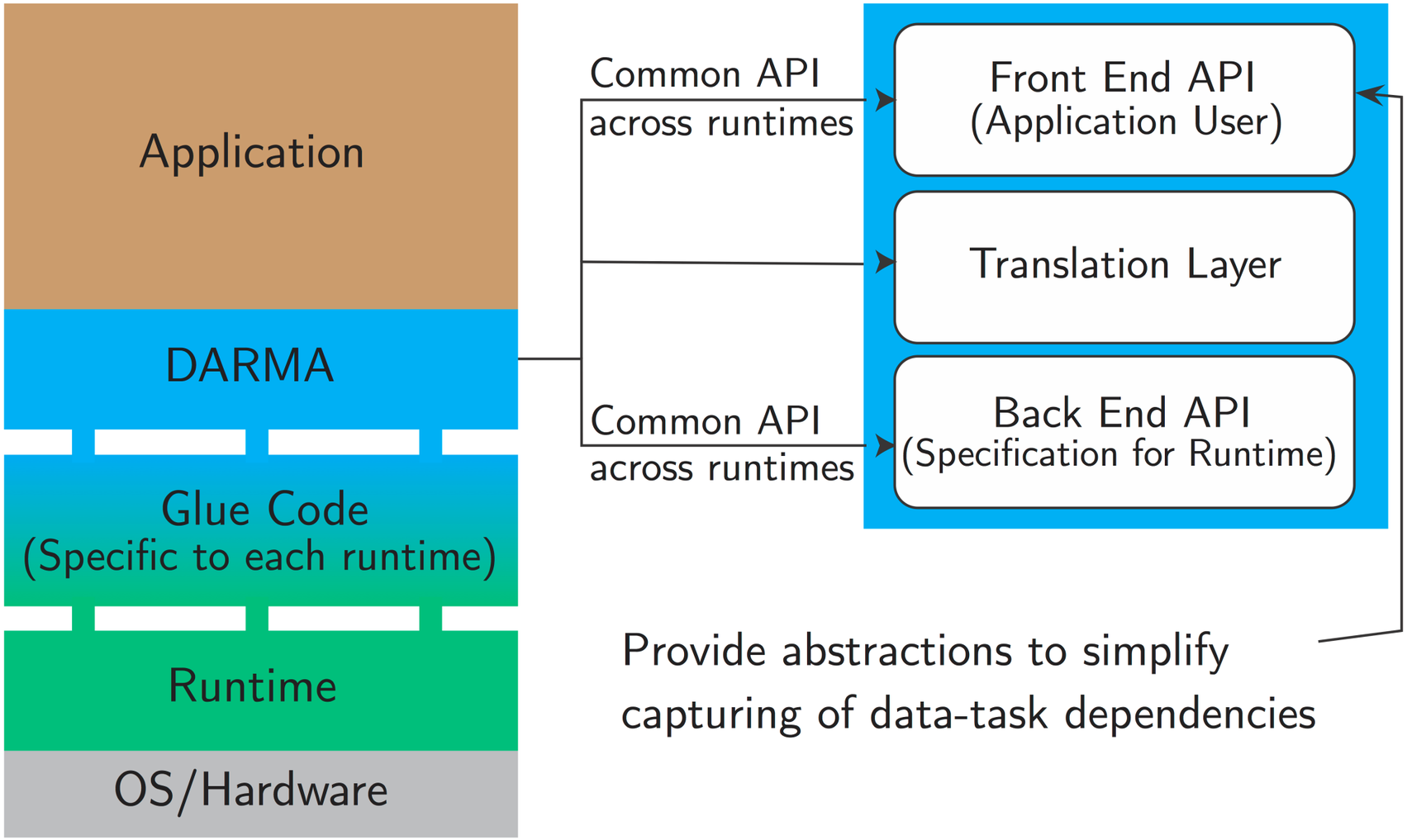
- `mpass::async()` detects dependencies of a task and their use (i.e., read or modify).
- Concurrency with other tasks is implicitly specified by how the data is used
- For simplicity, `mpass::async()` does **not** have a return value.
- A backend task scheduler and runtime layer is needed to execute the DAG.

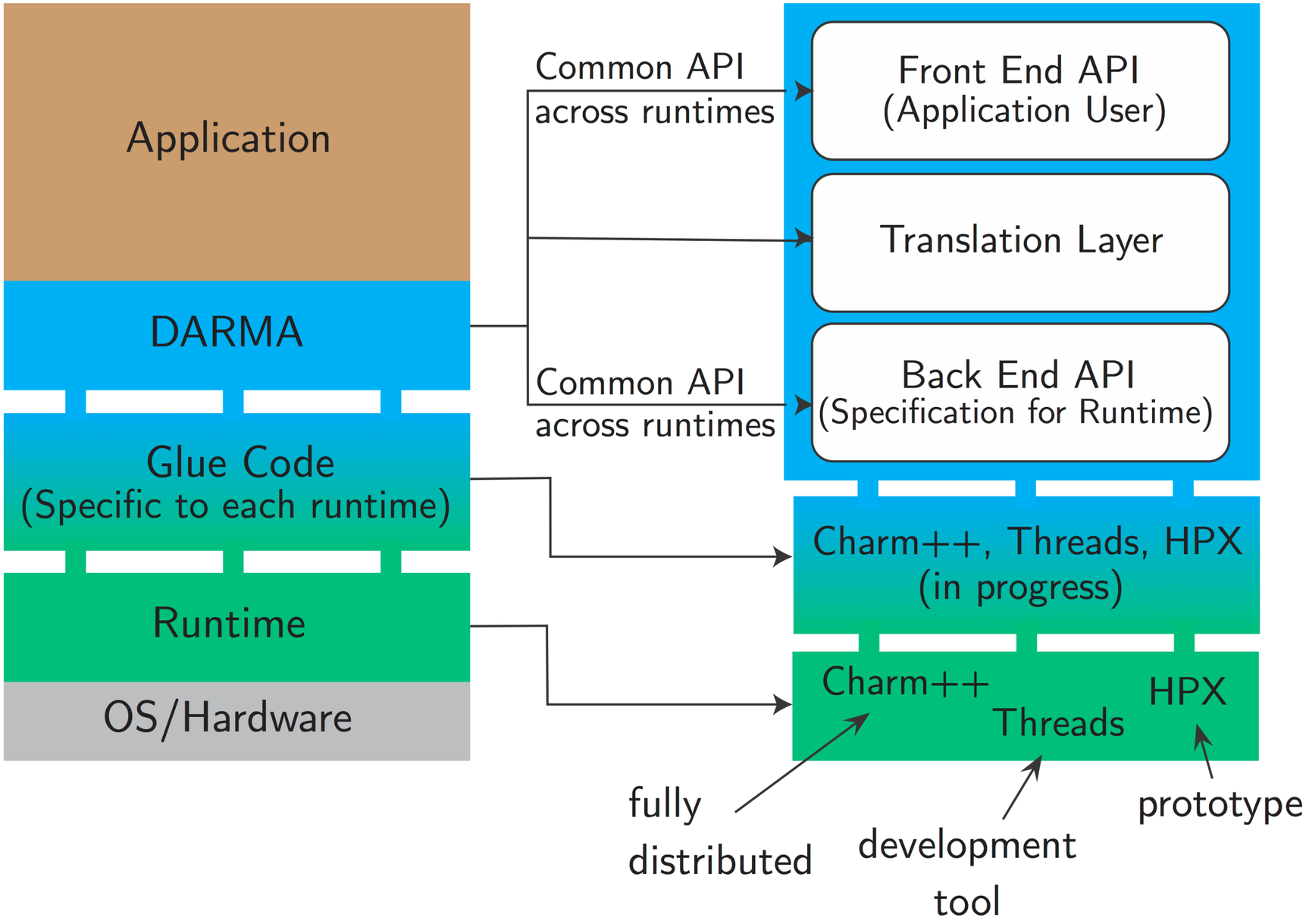

App in DARMA
Sequential Semantics
A
B
C
D
E
F


App in DARMA
Sequential Semantics
A
B
C
D
E
F
DARMA
App in a runtime
A
B
C
D
E
F


App in DARMA
Sequential Semantics
A
B
C
D
E
F
DARMA
App in a runtime
A
B
C
D
E
F
Runtime
App on a hardware
Time
node 1
node 2
C
A
E
D
B
F


Application
DARMA
Glue Code
(Specific to each runtime)
Runtime
OS/Hardware
Common API
across runtimes
Common API
across runtimes
Front End API
(Application User)
Translation Layer
Back End API
(Specification for Runtime)
Captures data-task
dependency information
Runtime controls construction and
execution of the DAG


Application
DARMA
Glue Code
(Specific to each runtime)
Runtime
OS/Hardware
Common API
across runtimes
Common API
across runtimes
Front End API
(Application User)
Translation Layer
Back End API
(Specification for Runtime)
Provide abstractions to simplify
capturing of data-task dependencies


Application
DARMA
Glue Code
(Specific to each runtime)
Runtime
OS/Hardware
Common API
across runtimes
Common API
across runtimes
Front End API
(Application User)
Translation Layer
Back End API
(Specification for Runtime)
Charm++, Threads, HPX
(in progress)
Charm++
Threads
HPX
fully
distributed
development
tool
prototype

How are data collections/data structures described?

- Asynchronous smart pointers wrap application data
  - Track meta-data used to build and annotate the DAG
    - Currently permissions information
    - Subsetting information under development

  How are data partitioning and distribution expressed?
- There is an explicit, hierarchical, logical decomposition of data
  - `AccessHandle<T>`
    - Does not span multiple memory spaces
    - Must be serialized to be transferred between memory spaces
  - `AccessHandleCollection<T,R>`
    - Expresses a collection of data
    - Can be mapped across memory spaces in a scalable manner
- Distribution of data is up to individual backend runtime

How is parallelism achieved?

- `create_work`
  - A task that doesn't span multiple execution spaces
  - Sequential semantics: the order and manner (e.g., read, write) in which data (`AccessHandle`) is used determines what tasks may be run in parallel
- `create_concurrent_work`
  - Scalable abstraction to launch across distributed systems
  - A collection of tasks that must make simultaneous forward progress
  - Sequential semantics supported across different task collections based on order and manner of `AccessHandleCollection` usage

How is synchronization expressed?

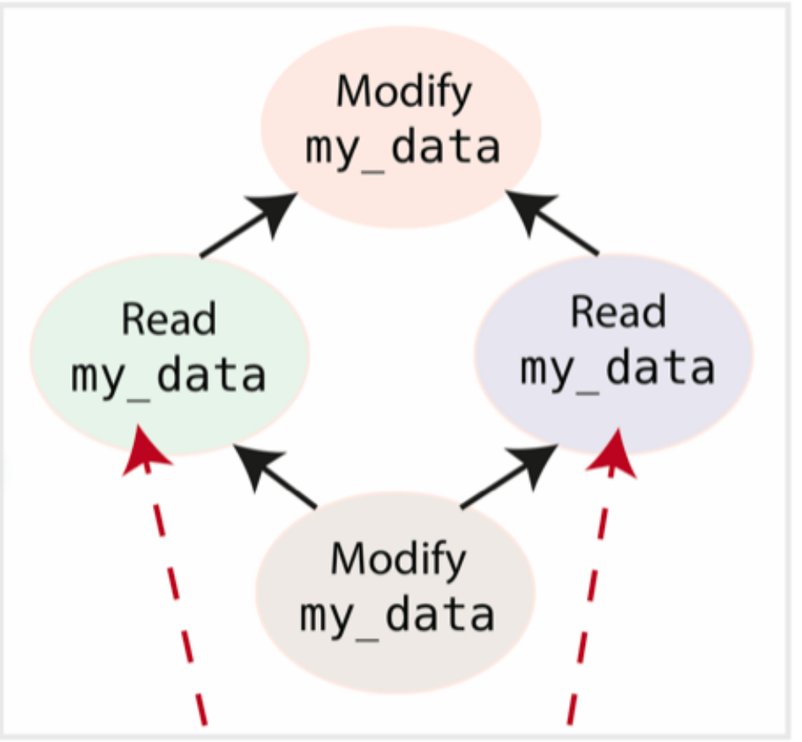
- DARMA does not provide explicit temporal synchronization abstractions
- DARMA does provide data coordination abstractions
  - publish/fetch semantics between participants in a task collection
  - Asynchronous collectives between participants in a task collection

Example Program

```
AccessHandle<int> my_data;

darma::create_work([=]{
  my_data.set_value(29);
});

darma::create_work(
  reads(my_data), [=]{
    cout << my_data.get_value();
  }
);

darma::create_work(
  reads(my_data), [=]{
    cout << my_data.get_value();
  }
);

darma::create_work([=]{
  my_data.set_value(31);
});
```

Sequential Semantics

DAG (Directed Acyclic Graph)



These two tasks are concurrent and can be run in parallel by a DARMA backend!

```
void darma_main_task(std::vector<std::string> args) {

  auto answer = initial_access<int>();

  //set value of answer - must run first
  create_work([=]{ *answer = 42; });

  //read-only, can run in parallel with check below
  create_work(reads(answer), [=]{
    std::cout << "The answer is" << *answer << std::endl;
  });

  //read-only, can run in parallel with print above
  create_work(reads(answer), [=]{
    if (*answer != 42){
      darma_runtime::abort("the answer is incorrect");
    }
  });

}
DARMA_REGISTER_TOP_LEVEL_FUNCTION(darma_main_task);
```

UQ with DARMA

- Uncertainties in inputs propagated to outputs:
  - Moments, reliability, PDFs of the outputs
- Techniques:
  - Sampling methods: ex. Monte Carlo, Multi-level MC, Importance sampling.
  - Functional expansion-based methods: ex. PCe.
- Need multiple evaluation of forward model (e.g. PDE).
- Why is DARMA (AMT) good for UQ?
  - (Dynamic) parallelism: heterogeneity among samples
  - AMT model is a natural fit
  - Nested UQ evaluations
  - Adaptive UQ algorithms
  - Performance portability, expressiveness and productivity

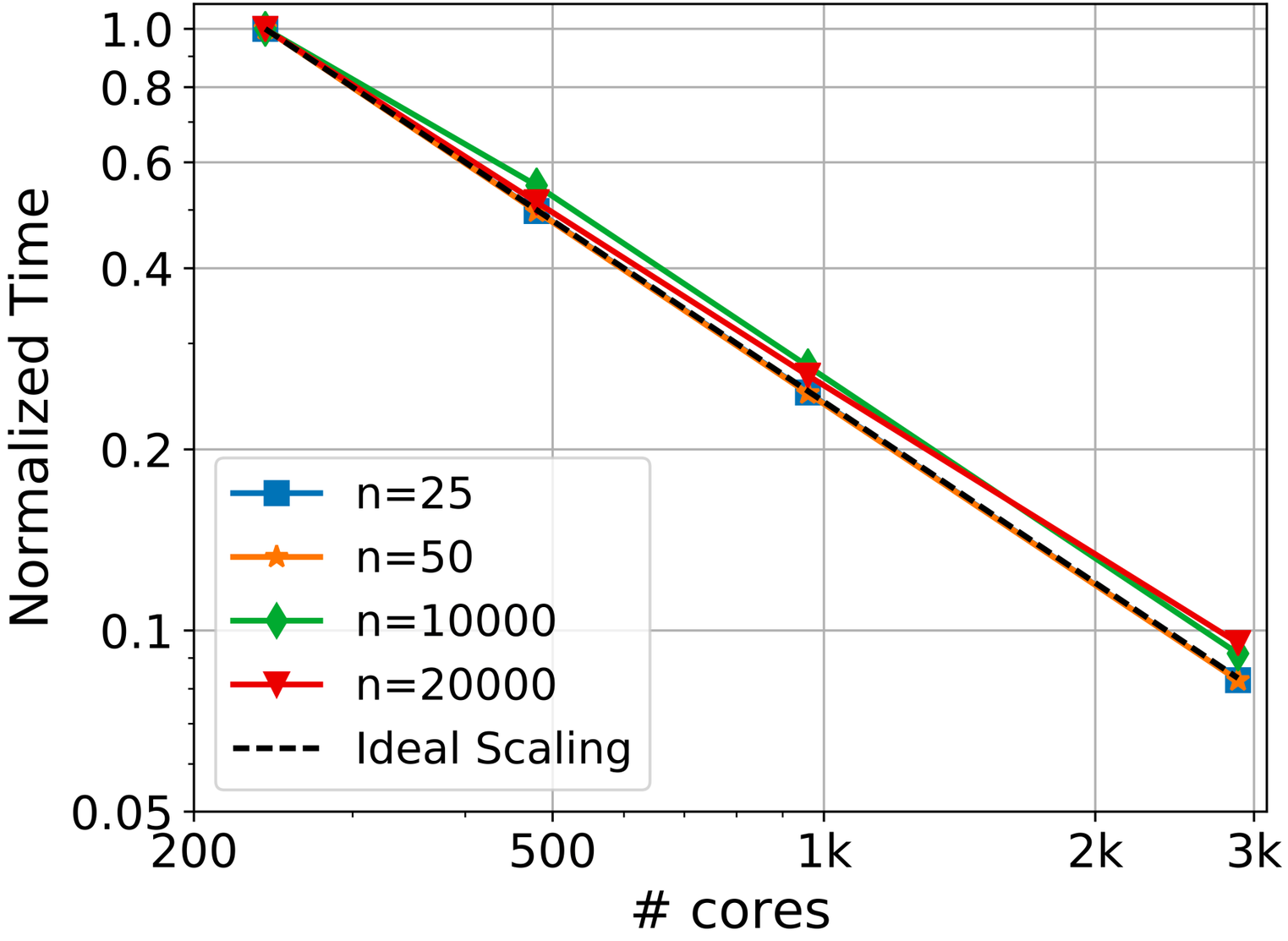
Two sample implementations

Multiple Solves per Rank

```
using vecD = vector<double>;

struct RunSamples {
  void operator()(
        Index1D<size_t> index, //...,
        AccessHandleCollection<vecD, Range1D> ahcdata) const
  {
    ahcdata[index].local_access().resize(solves_per_rank,0.0);

    for (uint i = 0; i < solves_per_rank; ++i){
      create_work([=]{
        // generate sample diffusivity
        // solve PDE for current germ sample
        // independently store QoI from this sample
      });
  }}
};
//
void darma_main_task(std::vector<std::string> args) {

const uint solves_per_rank = ...; //# of PDE solves per rank
const uint n_ranks         = ...; //# of ranks

auto data = initial_access_collection<vecD>(Range1D(n_ranks));

create_concurrent_work<RunSamples>(data, ..., Range1D(n_ranks));
create_concurrent_work<Collect>(data, ..., Range1D(n_ranks));
}
DARMA_REGISTER_TOP_LEVEL_FUNCTION(darma_main_task);
```

Single Solve per Rank

```
using vecD = vector<double>;

struct RunSamples {
  void operator()(
        Index1D<size_t> index, //...,
        AccessHandleCollection<double, Range1D> ahcdata) const
  {
    // generate sample diffusivity
    //...
    // solve PDE for current germ sample
    // store QoI from this sample
    //...

  }
};



void darma_main_task(std::vector<std::string> args) {


const uint n_ranks         = ...; //# of ranks

auto data = initial_access_collection<double>(Range1D(n_ranks));

create_concurrent_work<RunSamples>(data, ..., Range1D(n_ranks));
create_concurrent_work<Collect>(data, ..., Range1D(n_ranks));
}
DARMA_REGISTER_TOP_LEVEL_FUNCTION(darma_main_task);
```

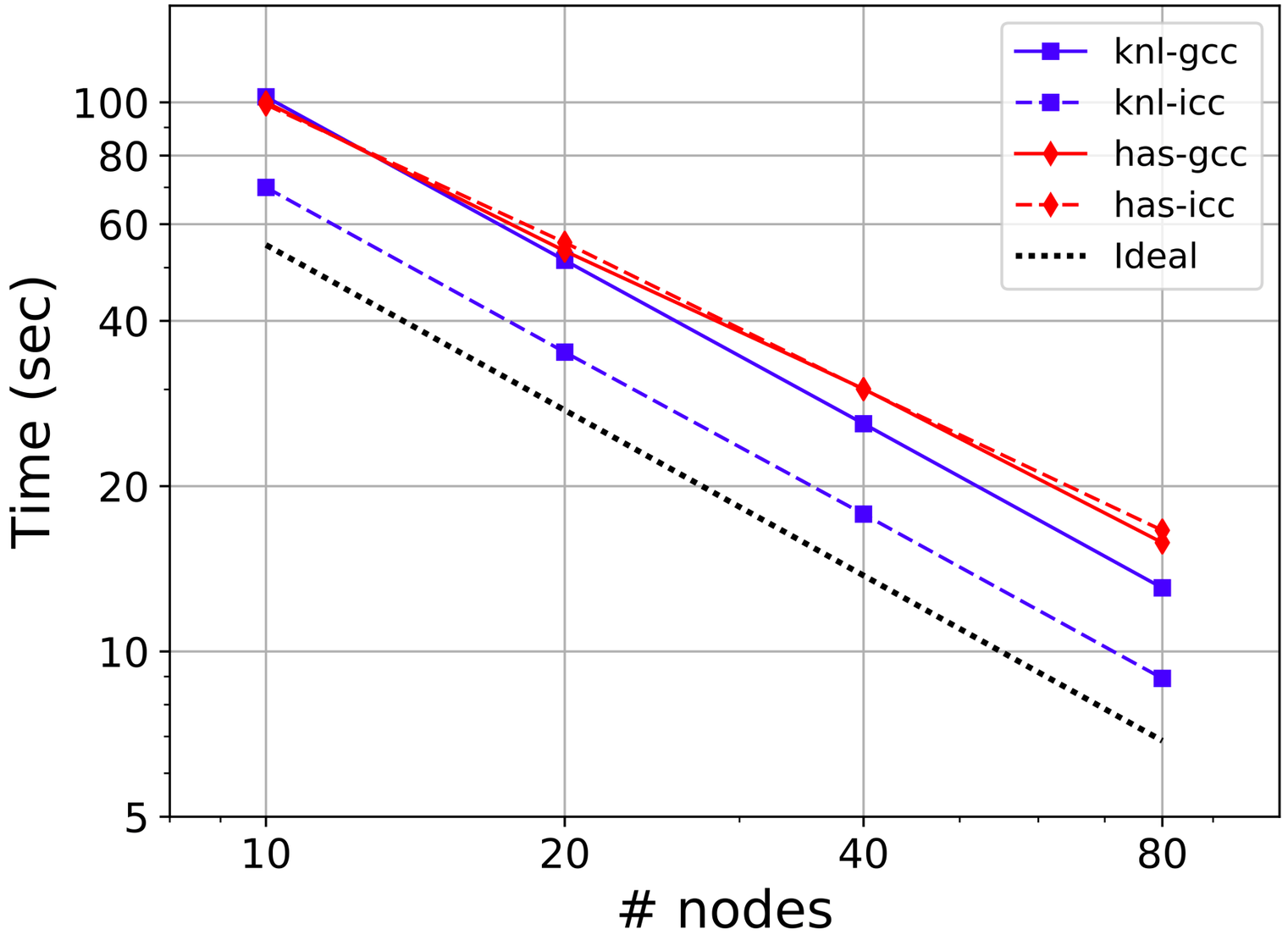

1.0
0.8
0.6
0.4
0.2
0.1
0.05
Normalized Time
200
500
1k
2k
3k
# cores
n=25
n=50
n=10000
n=20000
Ideal Scaling

Top-level Task

```
void darma_main_task(std::vector<std::string> args) {

auto vLevelsH = initial_accesss<vector<Level>>();
create_work<initialize>(vLevelsH, ...);

auto converged = initial_accesss<bool>();
auto iter = initial_accesss<uint>();
create_work([=]{
  converged.set_value(false); iter.set_value(1);
});

create_work_while([=]{
  return converged.get_value()==false && iter.get_value()<=maxIter;
}).do_([=]
{
  // Collect Samples
  collectSamples<runFunctor>(vLevelsH, ...);

  // compute stats, set new # of samples, check convergence
  create_work<checkStats>(vLevelsH, tolerance, converged, ...);
  iter.get_reference()++;
});

// compute estimator
create_work<MLestimator>(vLevelsH, mlmcEst);
create_work([=]{
  cout << " ML Value = " << mlmcEst.get_value() << endl;
});

}
DARMA_REGISTER_TOP_LEVEL_FUNCTION(darma_main_task);
```

Core Function

```
struct runFunctor{
  void operator()(
    Index1D index, uint iteration,
    int l, int l_offset, uint N,
    AccessHandleCollection<Storage> storeAHC, ...) const
  {
    const auto contextSize = index.max_value + 1;
    auto storage_h = storeAHC[index].local_access();

    uint myN = std::ceil(N/contextSize);
    for (uint i = 0; i < myN; ++i){

      create_work([=]
      {
        // generate sample of stochastic diffusivity
        create_work([=]{
          // PDE solve for l level (fine)
        });
        create_work([=]{
          // PDE solve for l-1 level (coarser)
        });

        // store target QoI for fine Q_l
        // store target QoI for coarse Q_lm1
        // store target QoI: Y = Q_l - Q_lm1;
      });
  }}
};
}
```


knl-gcc
knl-icc
has-gcc
has-icc
Ideal
Time (sec)
100
80
60
40
20
10
5
10
20
40
80
nodes

- Leverage data reusability (in progress):
  - Reuse data produced by some tasks to accelerate convergence for other similar tasks.
  - Tradeoff between data movement and execution time.
- Benchmarking for distributed backends
- Optimize load balancing methods for UQ applications

- https://share-ng.sandia.gov/darma/
- Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy’s National Nuclear Security Administration under contract DE-NA0003525.

Questions? Comments?

Thank you for your attention!