HiHAT: A New Way Forward

for Hierarchical Heterogeneous Asynchronous Tasking

A retargetable interface for tasking & language runtimes

CJ Newburn, Sean Treichler, Max Grossman,
Vincent Cave, Stephen Jones, James Beyer

Version 170515.1800

MOTIVATION FOR RETARGETABLE INFRASTRUCTURE

Build it right, for lasting impact

- "We haven't agreed on a user-level interface for tasking"
  - It's unlikely that we will anytime soon. But we can agree on infrastructure.
- "We're done with science experiments and want something we can use"
  - Gather usage models and requirements → architect a durable, robust solution
- "We don't want another academic endeavor"
  - Create something driven and supported by vendors

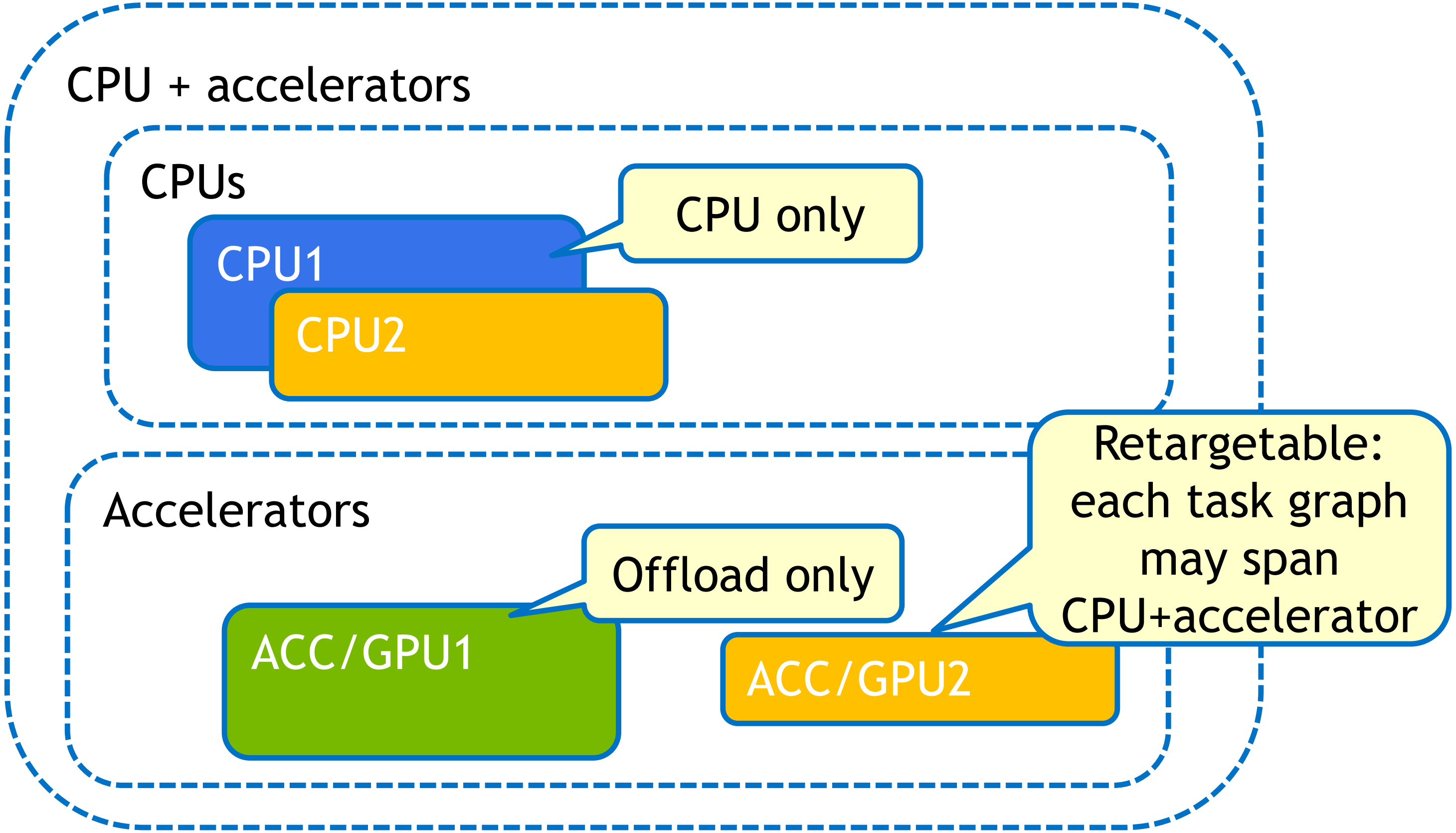
WHAT’S IN IT FOR THE COMMUNITY?

Seeking a win-win-win

- App developers
  - Common SW architecture across multiple targets
- Runtime developers
  - Better performance and robustness, less effort
  - Tasking runtimes and language runtimes that don’t necessarily use tasking
- Vendors
  - Expose HW features to a larger market, i.e. SW that spans multiple targets

COMMON SW ARCHITECTURE FOR HETEROGENEOUS SYSTEMS

Enables SW portability, broadens market for each vendor



WHERE DO WE START?

Bottom up

- In order to make life easier for the largest set of people, start at the bottom
    - Extremely performant APIs that span targets, plus an easier-to-use set of APIs
    - Strive for inclusiveness and extensibility
- Progress from low-level plumbing to runtime building blocks
    - Building blocks or anything higher are useless until you have underlying plumbing layer
    - Foster collaboration once we have something to work off of
- Make it easy to create new or improved user interfaces
    - But don’t start by convincing anything to quit using their and use a new user interface

WHAT WOULD IT NEED FOR BROAD ADOPTION?

Top down and bottom up: hihat

- Has to meet all provisioning constraints – see list below
- Has to be performant and robust and extensible – see design below
- Has to be the easiest way to get what people want – incremental, meeting needs
- Has to be driven by vendors, who are incentivized to be successful

Play the constructive skeptic today, and make your own evaluation

HW VENDORS AT MINI-SUMMIT

- Varying degrees of involvement. Some part of each company has expressed interest.

- AMD – **Ashwin Aji**, Mike Chu
- ARM – **Pasha Shamis**, Geraint North
- Cray – **Adrian Tate** (posted, was unavailable)
- IBM – **Wang Chen**, Kathryn O’Brien, Sean Nijjar
- Intel – **Vincent Cave**, Josh Fryman, Bala Seshasayee
- NVIDIA – **CJ Newburn, Sean Treichler, Stephen Jones, James Beyer**

- Host: **Wilf Pinfold**, Modelado.org, a neutral party funded by vendors and others

LANGUAGE OR TASKING FRAMEWORKS

- Varying degrees of involvement. Those in bold posted presentation materials.
- Some part of each has expressed interest.
  - C++ (**CodePlay**, IBM)
  - CHARM++ (**UIUC**)
  - Darma (**Sandia**)
  - Exa-Tensor (**ORNL**)
  - Fortran (IBM)
  - Gridtools (CSCS, Titech)
  - HAGGLE (PNNL/HIVE)
  - HPX (CSCS)
  - Kokkos, Task-DAG (**SNL**)
  - Legion/Realm (**Stanford/NV**)
  - OCR (**Intel**, **Rice**, GA Tech)
  - PaRSEC (**UTK**)
  - Raja (**LLNL**)
  - Rambutan, UPC++ (LBL)
  - R-Stream (**Reservoir Labs**)
  - SyCL (**CodePlay**)
  - SWIFT (**Durham**)
  - TensorRT (**NVIDIA**)
  - VMD (**UIUC**)
- Similarly with implementations
  - Argobots (**ANL**)
  - OpenCL (**CodePlay**)
  - Qthreads, NoRMa (**SNL**)
  - UCX/UCS (**ARM**)
- And a sampling of end users
  - ANL, **ANSYS**, Blue Brain (EPFL), CSCS, ECP, GROMACS (KTH), ICIS.PCZ.PL, Lattice Microbes (UIUC),LANL, LBL, **LLNL**, NEMO5 (Purdue), **ORNL**

AGENDA FOR MINI-SUMMIT

- Overview, motivation, status of HiHAT → common ground of understanding
- Review of vendor interest → growing level of support, more to go
- Review of runtime client interest → first set of viable customers
- Design principles and directions → deeper understanding
- Gather feedback → direction check

WHAT IS HiHAT?

4 faces

Community-wide **requirements** gathering effort

- Leads to solid architecture that's durable, extensible, robust

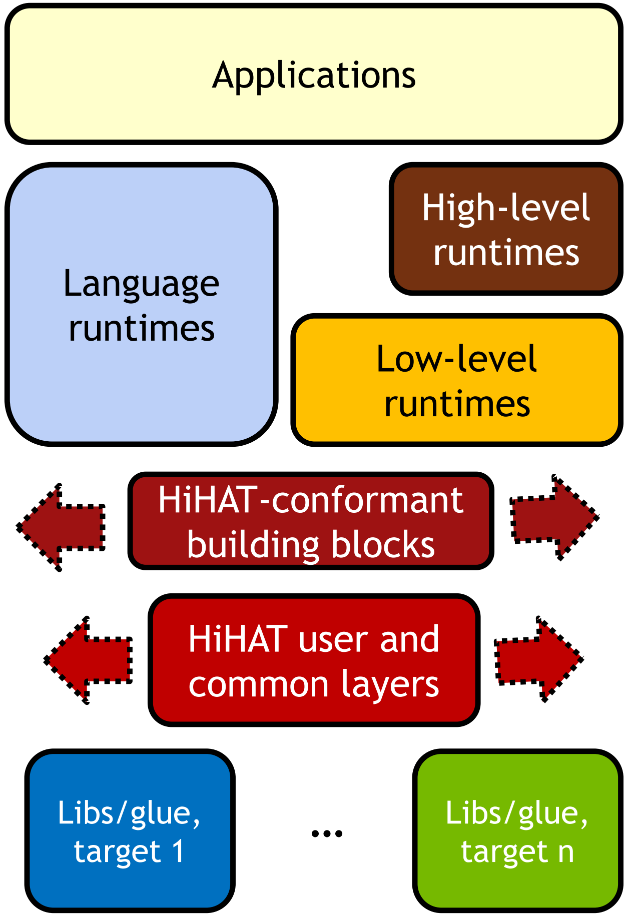
Architect user layer and common layer **API and implementation**

Implementation **beneath** user and common layers

- Vendor-maintained and user-supplemented

Integrate with OSS project: pluggable, conformant **building blocks**

- Built on user and common layers
- Language and tasking runtimes are built out of these



Applications

Language runtimes

High-level runtimes

Low-level runtimes

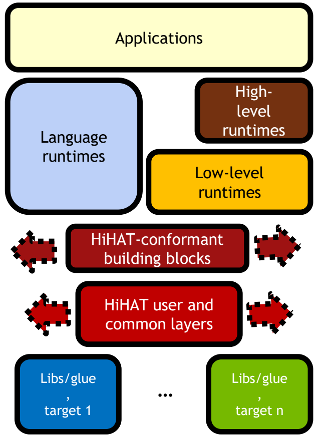
HiHAT-conformant building blocks

HiHAT user and common layers

Libs/glue, target 1

…

Libs/glue, target n



HiHAT CLIENTS

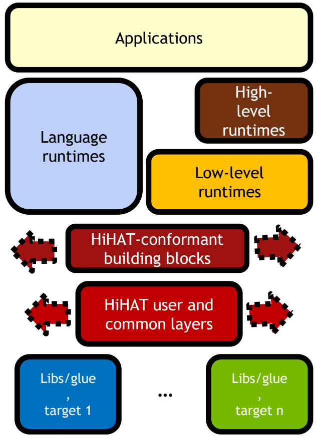
Start incrementally, build from there

- HiHAT's primary clients are **existing** language and tasking runtimes (e.g. C++, Kokkos)
  - Already have an interface to 1 or more targets, want a better interface/implementation
- HiHAT's secondary clients are runtimes that are **being designed** (e.g. HIVE/HAGGLE)
  - Open to influencing their design to be amenable to integration with/building on HiHAT

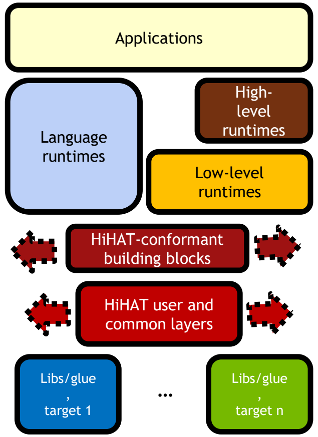
- HiHAT provides a **target-neutral** interface, used **whole or in part** by clients
  - Identify what's of greatest value, e.g. for future proofing, ease, robustness
  - **Incrementally adopt** those **parts of HiHAT**, and build up and out from there
- HiHAT does not have a near-term goal of providing a complete user-facing runtime

HiHAT'S OSS BUILDING BLOCKS

Accelerating communal progress



- The HiHAT interfaces will define types and a machine model
  - This HiHAT definition defines an architecture to which clients built on it conform
- Clients sharing a need for common functionality contribute/use building blocks
  - This would be an open source project
  - Examples: schedulers, cost models, visualization, dependence analysis, transformation
  - Suppose 4 orgs have needs in common; each can contribute a couple, consume others
  - Contributors can share tests (unit, functional, longevity)
  - Consumers can customize and contribute back, beef up testing, etc.

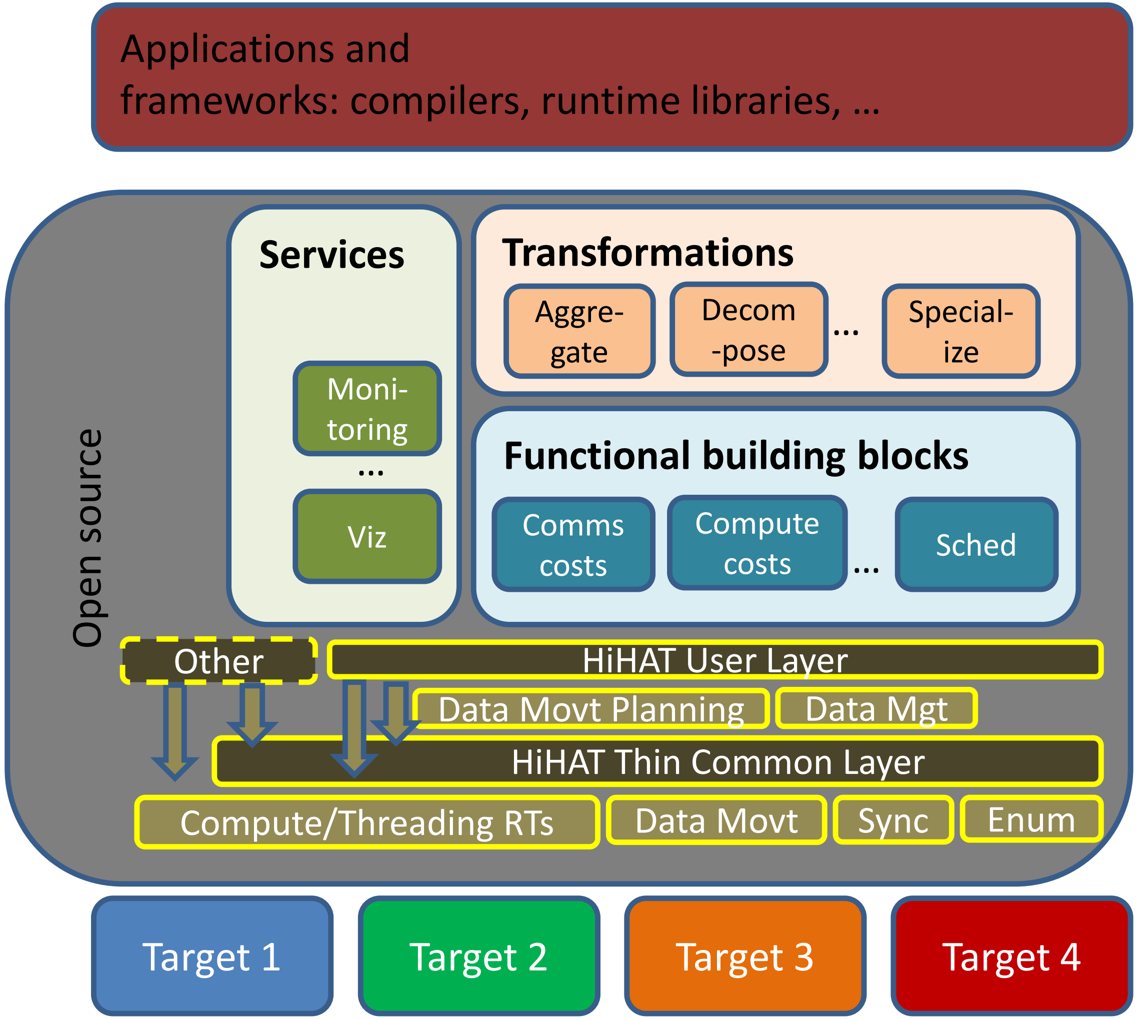


HiHAT'S IMPLEMENTATION LAYER

Vendor-driven performance and completeness

- HiHAT enables vendors/implementation providers to plug in functionality from below
  - Functionality behind the HiHAT APIs
  - Vendors may have the strongest incentive to provide access to their platform features
  - Others may offer alternate/improvements implementations

Layer	Value
Many frameworks	Many hats
Shared, contributed utilities	Accelerate coding Share technology Increase robustness
Target agnostic Target specific	Increase robustness More portable, tunable
Targets	Future proofing



Applications and frameworks: compilers, runtime libraries, …

Open source

Services

Moni-toring

…

Viz

Transformations

Aggre-gate

Decom-pose

…

Special-ize

Functional building blocks

Comms costs

Compute costs

…

Sched

Other

HiHAT User Layer

Data Movt Planning

Data Mgt

HiHAT Thin Common Layer

Compute/Threading RTs

Data Movt

Sync

Enum

Target 1

Target 2

Target 3

Target 4

https://wiki.modelado.org/HiHAT_SW_Stack

VALUE

Providing the easiest path toward what you already want

- Common interface to vendor-specific features
  - Modular design, separation of concerns
    - What's above user/common layer can use target-agnostic heuristics on target-specific parameters
  - Future proofing
    - Retargetable across vendors, implementations, generations
    - Underlying implementations can chase changes and improvements
- Performance and robustness
  - Vendors are incentivized to provide 1st-class support; others can supplement

STATUS

Gradual start, but on firm footing

- Gather
  - Usage models, applications, user requirements – modestly-broad participation, need more
- Architect
  - Design principles – good progress, much more to come; need more concrete requirements
- Implement
  - Implementation plan – POC this summer, anticipating partial implementation end of 2017
- Integrate
  - Proof of concept → early adopters → broaden

Opt Timing	2016 | 2017 | 2018
Gather	Community input Community review Community feedback
Architect	Design principles API proposal Refined API Updated API
Implement	Proof of concept Initial subset More complete
Integrate	Proof of concept First/partial clients Broader, more complete

MOMENTUM

Building interest, firming up investment

- Modelado.org – neutral zone, posting of usages, requirements, apps; monthly mtgs
- Active bottom-up discussions with vendors → initial POC with glue code
- Existence proofs and past learning: hetero streams, REALM, ~OCR, CodePlay
- ECP – ATDM funding, PathForward2 SW, CORAL/APEX/ECP app owners from ORNL, ANL, LBL, LANL
- PASC – interest from Platform for Advanced Scientific Computing, Switzerland
- Workshop on Exascale SW Technologies (WEST) – panelist, Feb. 22
- Workshop at GPU Tech Conference – May 9 am, share progress, deepen investment
- Talk @ IWOCL workshop, Distributed and Hetero Programming for C/C++17, May 16
- Performance portability workshop – August 21

A RETARGETABLE FRAMEWORK

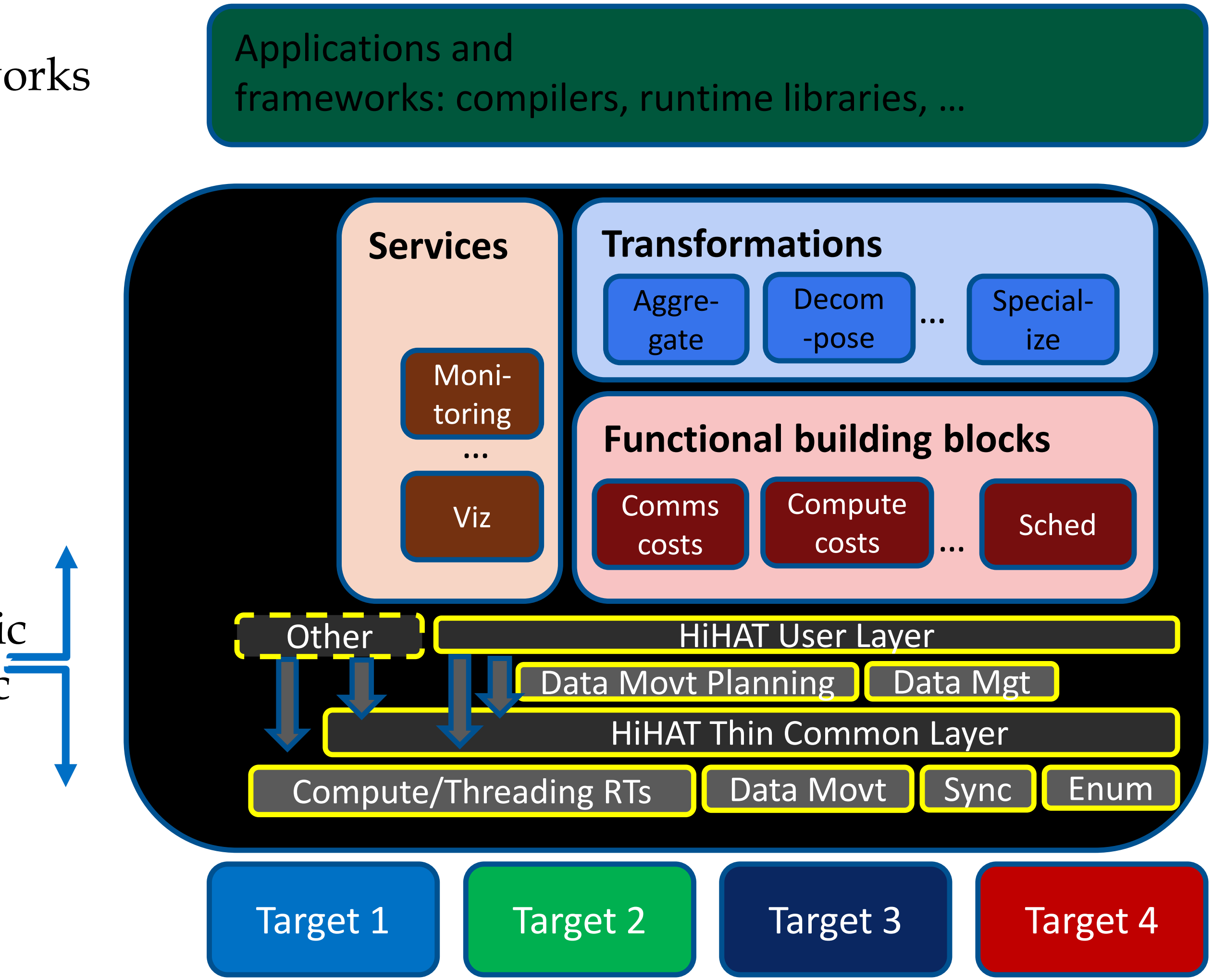
Interfaces are common across multiple targets

Action / service	Description	Example
Computation	Target-specific code isolated in tasks, different implementation for each target, layout	*Invoke task named FOO on target X*
Data layout	Multiple data layouts, with implementations specialized for each	*Data layout Y (vs. Z) is used to select which implemention of FOO to execute*
Data movement	Bring input data for task T to where it executes, send output data to where it's needed	*Fetch FOO's data from wherever it was produced, send its outputs to consumers* *Optionally re-layout data on the way*
Coordination	Observe and enforce data and control dependences	*FOO doesn't execute until its predecessors complete, the data is sent and formatted*
Scheduling	Select best resources to bind computes and data to and ordering, based on cost models Trade-off across multiple targets, data layouts	*FOO doesn't execute until its predecessors complete, the data is sent and formatted*

SCOPE OF FUNCTIONALITY

- Cover key platform-specific actions and services
  - Data movement – target-optimized copies, DMA, networking
  - Data management – support many kinds and layers of memory, specialized pools
  - Coordination – completion events, locks, queues, collectives, iterative patterns
  - Compute – target-optimized tasks, including remote invocation
  - Enumeration – kinds and number of resources (compute, memory), topologies
  - Feedback – profiling, load
  - Tools – tracing, callbacks, pausing, … {debugging}

Layer	Value
Many frameworks	Many hats
Shared, contributed utilities	Accelerate coding Share technology Increase robustness
Target agnostic Target specific	Increase robustness More portable, tunable
Targets	Future proofing



https://wiki.modelado.org/HiHAT_SW_Stack

Max

SAMPLE APPLICATIONS

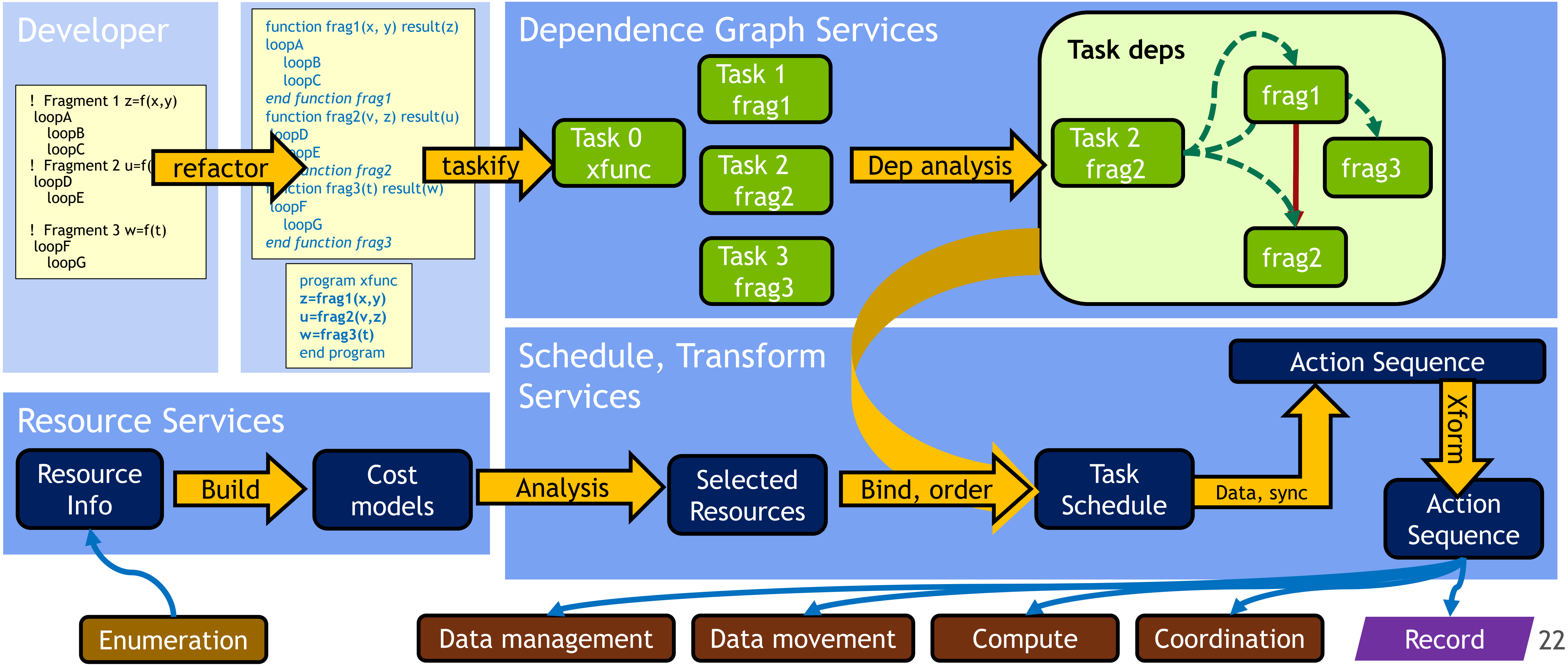
Usages that could benefit from HiHAT

- HiHAT is driven to solve real problems for real users (not a research project)
- Key early goal of this effort is to identify concrete applications and parallel patterns
- Small subset presented here:
    - Unbalanced Tree Search
    - Smith Waterman for genome alignment
    - Deep Learning kernels
    - Multipole
    - Monte Carlo Transport
    - ANSYS solvers on retargetable framework
    - DMRG++ nanomaterials

Vincent

SAMPLE FLOW

Mapping of what developer and services perform on top of HiHAT



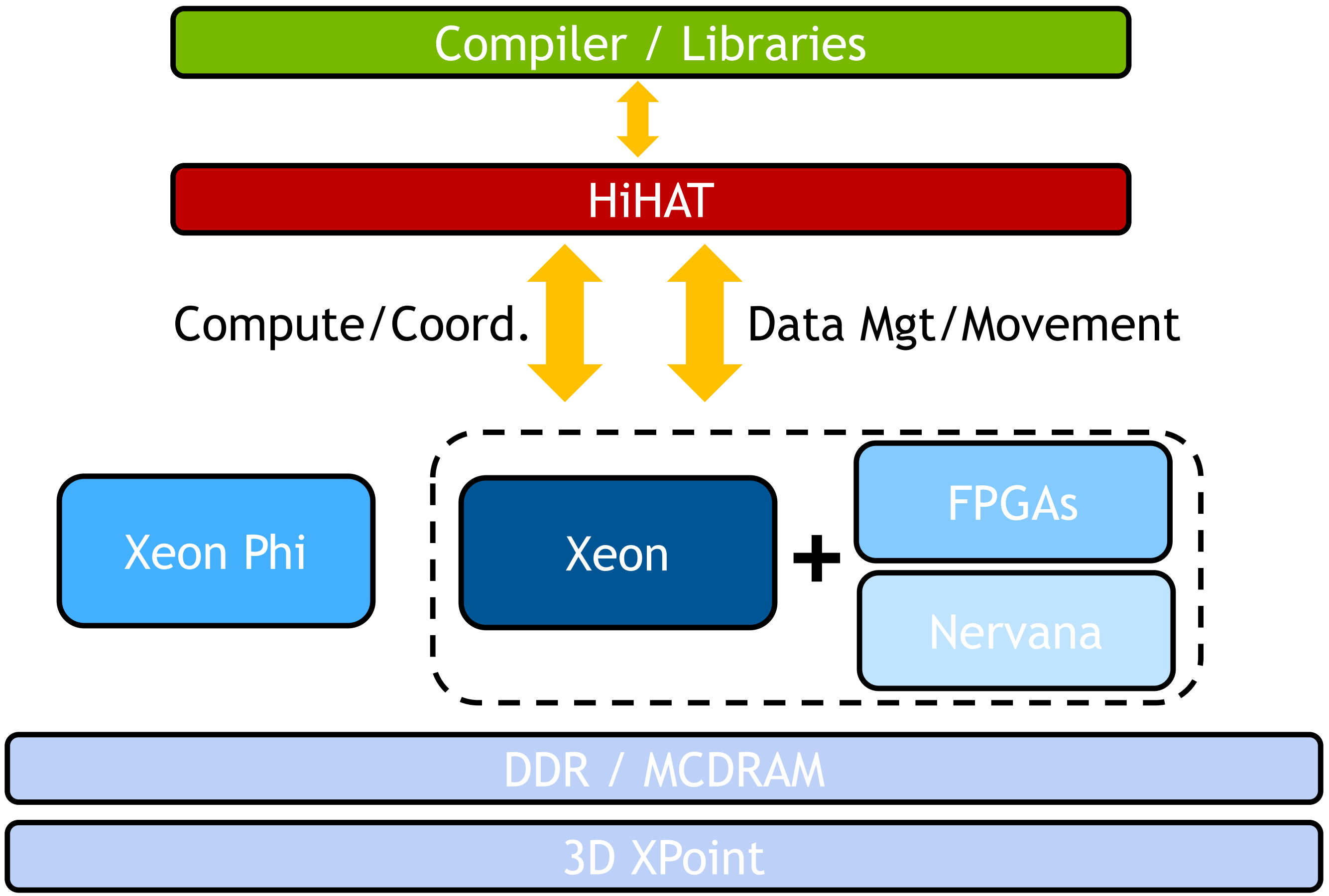
```
! Fragment 1 z=f(x,y)
 loopA
   loopB
   loopC
! Fragment 2 u=f(
 loopD
   loopE

! Fragment 3 w=f(t)
 loopF
   loopG
```

```
function frag1(x, y) result(z)
loopA
   loopB
   loopC
end function frag1
function frag2(v, z) result(u)
loopD
   loopE
end function frag2
function frag3(t) result(w)
loopF
   loopG
end function frag3
```

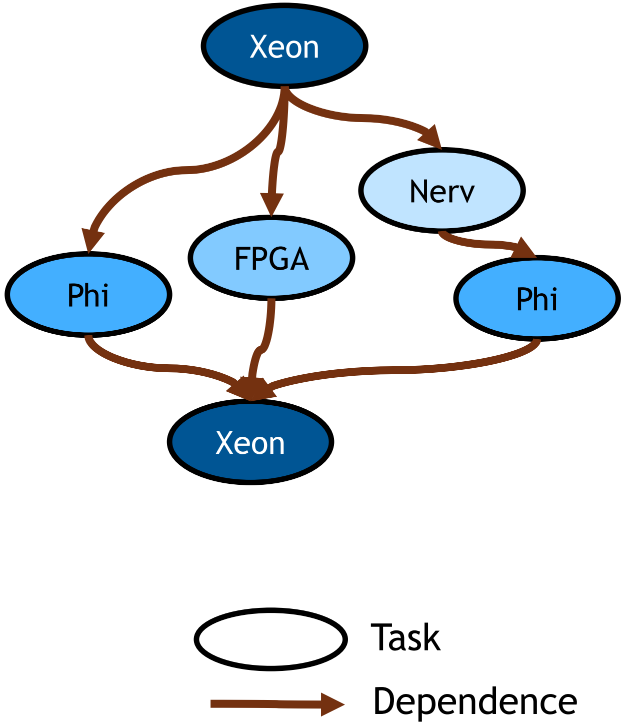
```
program xfunc
z=frag1(x,y)
u=frag2(v,z)
w=frag3(t)
end program
```

Vincent

SCHEDULING AND INVOCATION EXAMPLE

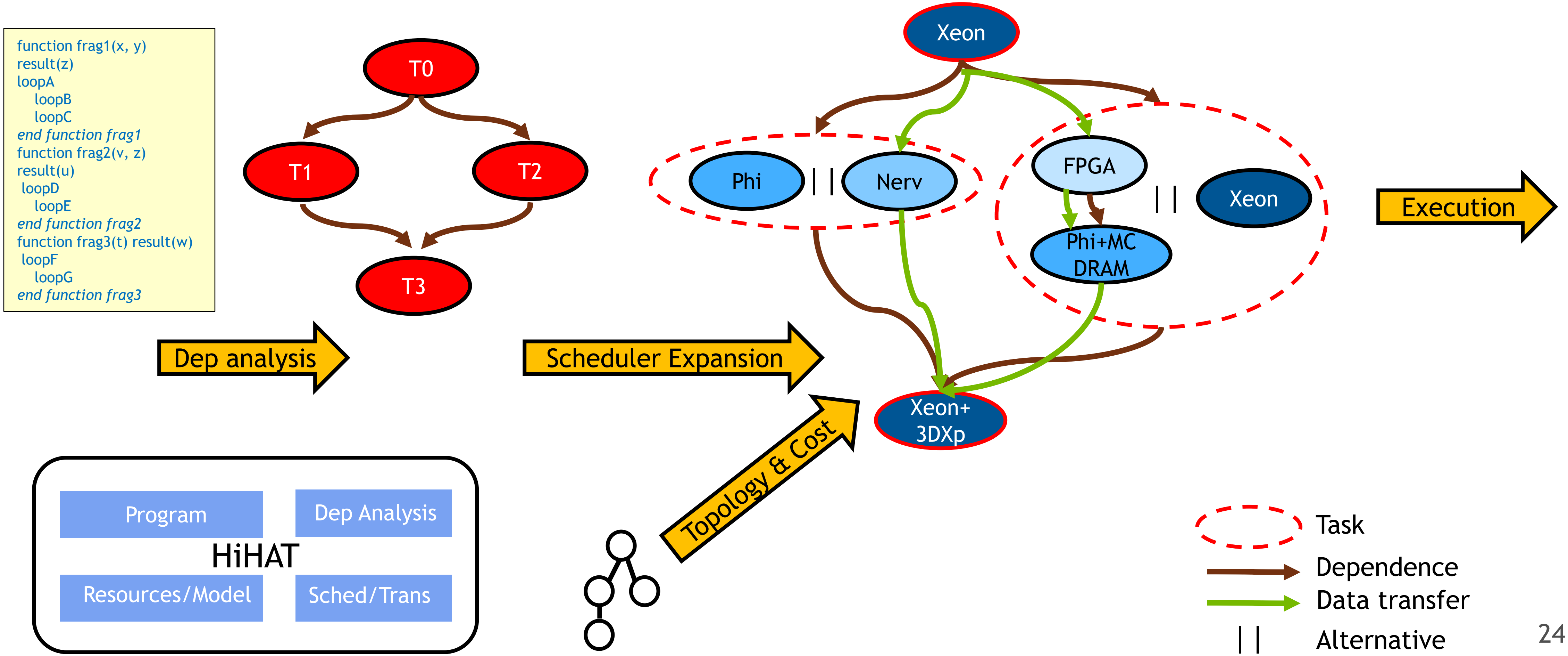


HiHAT task graph mapped to resources



Vincent

SCHEDULING AND INVOCATION EXAMPLE

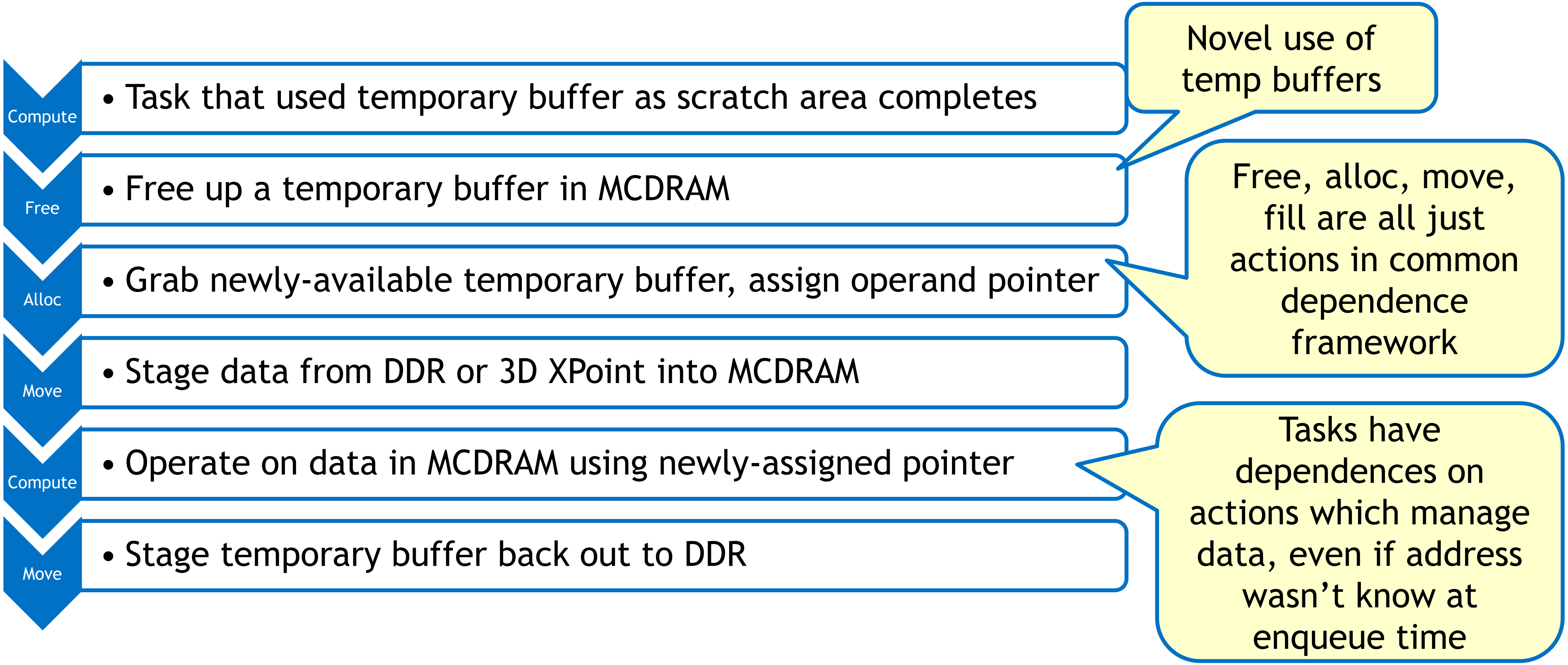


```
function frag1(x, y)
result(z)
loopA
    loopB
    loopC
end function frag1
function frag2(v, z)
result(u)
 loopD
    loopE
end function frag2
function frag3(t) result(w)
 loopF
    loopG
end function frag3
```

Vincent

DATA MANAGEMENT AND MOVEMENT

These are all actions that depend on one another



Compute
- Task that used temporary buffer as scratch area completes

Free
- Free up a temporary buffer in MCDRAM

Alloc
- Grab newly-available temporary buffer, assign operand pointer

Move
- Stage data from DDR or 3D XPoint into MCDRAM

Compute
- Operate on data in MCDRAM using newly-assigned pointer

Move
- Stage temporary buffer back out to DDR

Novel use of temp buffers

Free, alloc, move, fill are all just actions in common dependence framework

Tasks have dependences on actions which manage data, even if address wasn’t know at enqueue time

DATA MOVEMENT EXAMPLE

Resolving the abstraction as you get close to the metal

- Input: Move a collection of 5K blocks of various sizes from {CPU, GPUs} to {CPU, GPUs}
- Aggregate: Bundle contiguous chunks to same target → fewer, larger chunks
- User layer <source, target, size>
  - Instance resolution*: find closest, latest copy of source; find target affinity
  - Alias detection*: nop-ify when source & target are aliased, but maintain transitive deps
  - Pick transport type: above size threshold → DMA ops, below threshold → memcpy ops
  - Pick transport type: best RDMA implementation for end points
  - Address mapping: adjust source/target addresses by appropriate offsets for their domain
- Common layer <source domain, source adr, target domain, target adr, size, type>
  - DMA: Invoke DMA on CPU or GPU, or RDMA to remote CPU/GPU
  - Memcpy: T-threaded memcpy for T-thread targets, cudaMemcpy

- **May be done above user layer*

VENDOR INTEREST

Some part of each institution has expressed technical interest, not necessarily business commitment.

- **Intel - Vincent Cave**, Josh Fryman, Bala Seshasayee
- **NVIDIA – Stephen Jones**, CJ Newburn, Sean Treichler, James Beyer
- **AMD –Ashwin Aji**, Mike Chu
- **IBM – Wang Chen** (spoke, didn’t present)**,** Kathryn O’Brien
- **Cray - Adrian Tate** (remote → wasn’t available)
- **ARM – David Lacomber** (didn’t present)

IMPLEMENTATION LAYER INTEREST

Some part of each institution has expressed technical interest, not necessarily business commitment.

- **Argobots: Halim Amer, ANL**
- **Qthreads, NoRMa: Stephen Olivier, Sandia**
- **UCX/UCS: Pasha Sharmis, ARM (remote)**
- **SYCL/ComputeCPP: Michael Wong, Codeplay, Khronos, HSA (remote)**
- **C++: Michael Wong, ISOCPP (remote)**

LANGUAGE OR TASKING FRAMEWORKS

Some part of each institution has expressed technical interest, not necessarily business commitment.

- C++ (**CodePlay**, IBM) Michael Wong
- Charm++ (**UIUC**) Ronak Buch, (**Charmworks**) Phil Miller
- Darma (**Sandia**) Janine Bennett
- Exa-Tensor (**ORNL**) Wayne Joubert
- Fortran (IBM)
- Gridtools (**CSCS**, Titech) Mauro Bianco
- HAGGLE (**PNNL/HIVE**) Antonino Tomeo
- HPX (CSCS)
- Kokkos, Task-DAG (**SNL**) Carter Edwards
- Legion (**Stanford/NV**) Mike Bauer
- OmpSs (BSC) Jesus Labarta
- Realm (**Stanford/NV**) Sean Treichler
- OCR (**Intel**, **Rice**, GA Tech) Vincent Cave
- PaRSEC (**UTK**) George Bosilca
- Raja (**LLNL**) Rich Hornung
- Rambutan, UPC++ (LBL) Cy Chan
- R-Stream **(Reservoir Labs)** Rich Lethin
- SyCL (**CodePlay**) Michael Wong
- SWIFT (**Durham**) Matthieu Schaller
- TensorRT (**NVIDIA**) Dilip Sequeira
- VMD (**UIUC**) John Stone

TABULATED RESULTS

Strong interest, modestly amenable; focus on data first

Type of functionality	Level of interest			Amenability to refactoring		
	H	M	L	H	M	L
Data movement – target-optimized copies, DMA, networking	15	0	1	8	3	1
Data management – kinds and layers of memory, specialized pools	10	4	2	8	2	2
Coordination – completion events, locks, queues, collectives, iteration	9	7	0	6	4	1
Compute – local or remote invocation	7	2	4	4	5	3
Enumeration – kinds/# of resources, topologies	11	3	2	4	4	2
Feedback – profiling, utilization	6	5	3	4	6	1
Tools – tracing, callbacks, pausing, debugging	3	10	3	2	6	2

DESIGN PRINCIPLES

- Key provisioning constraints
- Architecture
  - HiHAT common and user layer only dispatch
  - Common layer for minimal overhead, user layer for ease of use
  - Interaction with network, memory models
- API design

PROVISIONING CONSTRAINTS

If the interface doesn’t support these, it may not be useful

- Must be options for very fast and reasonably usable
- Library that’s applicable to runtimes or static compilation
- C ABI that supports layering of C++, Fortran, Python on top
- Does not have to own main
- Incremental, does not have to own all memory allocation
- Composable and interoperable, e.g. with OpenMP, Kokkos, MPI, HDF5
- Can target heterogeneous systems, e.g. Xeon, Xeon Phi, GPUs, FPGAs, TPUs, ARM
- Tasks are spawnable and can sleep, e.g. can stall on IO, deschedule and resume
- Data movement should be extensible for network and relaxed memory
- Well-defined thread safety model (work through implications for user-level threads)
- Enough control to provide numerical reproducibility

COMMON AND USER LAYER: DISPATCH ONLY

Open, pluggable, low-overhead

- Primary Common Layer and User Layer APIs only do dispatch
  - HiHAT implementation is thin and light, easy to code, robust
  - Performant (TBD whether it's inlinable)
- Target-kind descriptor will be used to select target function to invoke
  - Register target-specific implementation under each API
  - Suppliers can be a mix of vendors and other implementation providers

LAYERING

Layer	Examples	Target handling	Implemented by	Functionality
Runtime	TensorRT, Legion, Kokkos, PaRSEC, Raja, C++ runtime, offload runtime	Target-agnostic implementation that may use target-specific info	Tuners	Make decisions, apply transformations, call services
Reusable modules	Dependence analysis, cost models, scheduler		Tuners, open sourced	Any kind of service that is commonly used and/or sharable
User layer	Configuration, data movem’t(logical source, log dest, size, layout), data mgt, invocation, sync	Map from target-neutral API to target-specific implementation	Target ninjas	Some decisions, can take longer, some overhead
Common layer	Data movem’t (source virtual address, dest VA, DMA/memcpy), data mgt, invocation, sync		Target ninjas	No decisions, absolutely minimal overhead

COMMON LAYER VS. USER LAYER

- HiHAT User Layer – logical to low-level mapping
  - Sample inputs for higher-level and configuration actions
    - < logical source, logical target, size, [descriptor,] completion event> or
    - <func_name, logical operands, input deps, completion event, flavor>
  - Action: Low-level operands: domain, low-level addresses
  - Characteristics: lookups, decisions
- HiHAT Thin Common Layer - function mapping only
  - Sample inputs for low-level operational actions
    - < Low-level operands, size, type, completion event> or
    - <func_name, low-level operands, input deps, completion event, flavor>
  - Action: invoke best-available implementation for that source [and target] domain
  - Characteristics: Razor thin, minimal overhead, no decisions
- Provide completion events

COMMON LAYER – THIN AND LIGHT

Many possible 3rd-party implementations to select from

Function	CPU	GPU	FPGA, …
Compute, threading	pthreads, OpenMP, Argobots, Qthreads	cu* library calls, CUDA kernels, OpenACC kernels	
Data movement	MPI, SHMEM, UCX, memcpy, DMA, GASnet	MPI/GPUDirect, nvSHMEM, cudaMemcpy, DMA, GASnet	
Synchronization and communication	MPI wait, MPI collectives	MPI collectives, NCCL, cudaEvent, …	
Data management	malloc, TBBmalloc, new, sbrk, mmap	cudaMalloc, cudaMallocManaged, {special pools}	
Enumeration	# cores, threads/core, ISA versions, hwloc, …	# devices, #SMs, compute version, topology, …	
Feedback	PAPI	PAPI, cupti	
Tools	Tracing? Callbacks?	Tracing? Callbacks?	

USER LAYER - THICKER AND RICHER

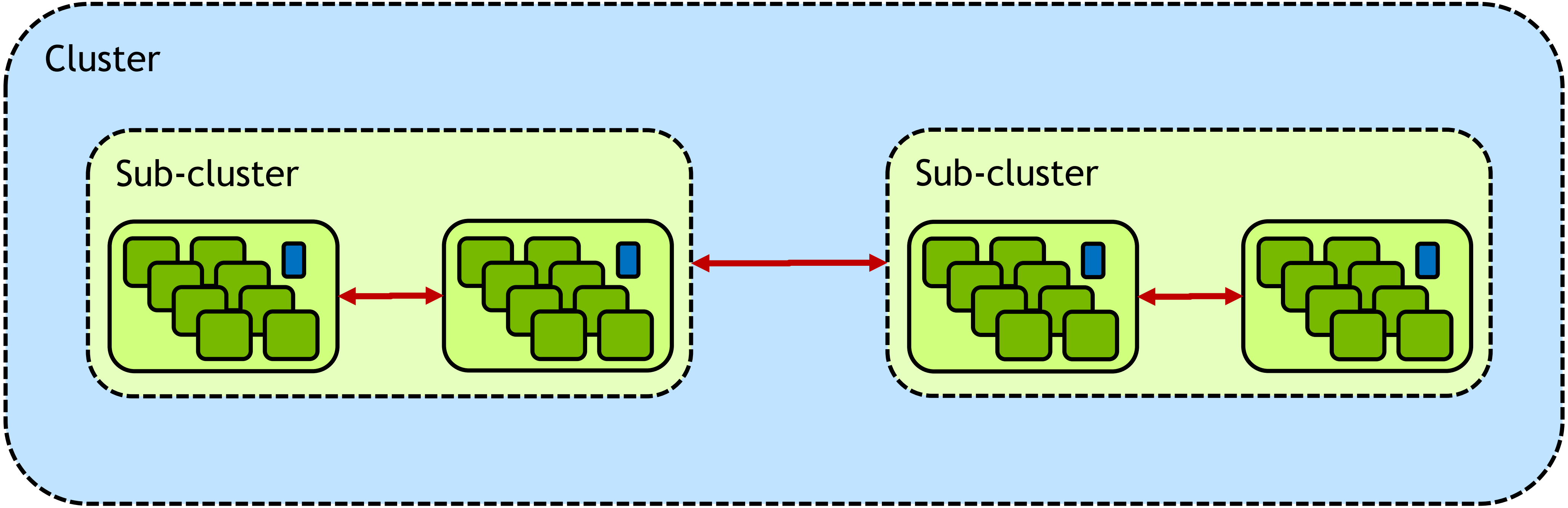
Some of these may set up later calls to the user or common layer

Function	CPU	GPU	FPGA, …
Compute, threading	Create OpenMP hot team, affinitize threads	Set default device	
Data movement	Choose transport mechanism, given endpoints and size	Choose transport mechanism, given endpoints and size	
Synchronization and communication			
Data management	Choose mem kind, allocator	Choose whether managed memory or not, choose cudaMemAdvise parameters	
Enumeration			
Feedback	Load indication	Load indication	
Tools	Debugging	Debugging, pause?	

TOPOLOGY

Domain (MPI rank/PE/process) managed by HiHAT → node/sub-cluster

Network may stretch within (sub-cluster) or among (nodes) domains



Feedback item

DATA MOVEMENT: NETWORK, MEM MODELS

Intra- and inter-domain data movement

- Composability with networking APIs, weak memory models is design criterion
  - Split phase, e.g. set up RDMA and use it
  - Split phase, e.g. write, fence/quiet
- HiHAT domain encapsulates some set of resources across which data movement can be expressed using only HiHAT APIs
  - Inter-domain compatibility with third-party communication libraries (e.g. MPI, SHMEM) beyond scope – support would have to be implementation dependent
  - HiHAT domain does not necessarily equal MPI rank (i.e. multi-node domains for neighborhood work-stealing)
- Inter-domain data movement is the responsibility of higher layers
  - Compose with these external actions through externally triggerable events
  - Consider composability for both relaxed models (SHMEM, UPC) and message passing
  - Separation of inter- and intra-domain communications doesn't necessarily avoid all resource contention

API DESIGN

- Varying levels of support: core, extended, metadata
- C ABI
- User function argument marshaling
- Granularity assumptions
- Inlining

Feedback item

VARYING LEVELS OF SUPPORT

API extensibility scheme supports a range from minimal to futuristic

- Core - essential
  - Must support to be conformant, goal for all targets
  - May have suboptimal performance, may map to a bundle of target-specific calls
- Extension - enhancements
  - May be supported by a subset of targets
  - Formalize the definition so that API captures most all semantics
  - Could extend core with performance, functionality, or ease of use
- Metadata – enable lower implementation layer
  - May be used for experimental purposes
  - May be target-specific blob

C ABI

- Generality: Fortran, C++, Python, …
- Ease of use: Easy to make fast, doesn’t require C++ expertise
- Provisioning constraint: Some customers prohibit the use of C++

James

Feedback item

USER FUNCTION ARGUMENT MARSHALING

This solution is applicable for C++, tasks, offloading (e.g. OpenMP)

- HiHAT client responsible for de/marshaling, including how to capture closure
  - Closure includes function and data environment (C/C++arguments and C++ data members)
  - Client may be compiler or programmer or library
  - Client can pass in demarshaling function or use some demarshaling convention
- What HiHAT has to support and deal with
  - No: considerations of operand size, type (type casting)
  - Yes: Accept pointer to function and pointer to data blob
  - Yes: Provide functions to convert pointers across domains, as needed

James

Feedback item

GRANULARITY ASSUMPTIONS

- Overhead determines supportable granularity
- Argument/environment marshaling is the client's responsibility
  - Could be largest cost be far, e.g. for multi-D, but could potentially be off of the critical path
- Length of code path
  - function calls to setup task – dependency setup/resolution, enqueue
  - target-based selection – this could include extra code marshaling
  - Call to start task – cost varies greatly depending on selection
- Execution speedup
- Achieved parallelism/concurrency

James

Feedback item

INLINING

- Motivation
  - Specialization – C++ templates
  - Simplification
  - Suitable for compiler
- Plausibility
  - 2 header versions: definition with “inline” or declaration only with definition in library
  - OpenACC: device_type specializes directive for different devices
  - Implementation is simple enough to effectively inline

Feedback item

ARCHITECTURAL CONFORMANCE IDEAS

Open: What are the key conformance requirements of building blocks?

- Tracing
- State
  - Bound: binding, bound, unbinding, unbound
  - Ready: not, partial, full
  - Execution: Pending, executing, completed successfully, error, canceled
- Debugging

MINI-SUMMIT FEEDBACK SUMMARY AND POLL

Discuss, then show of hands

Question	Proposed	Alternative	Alternative 2
HiHAT spans domains (processes/ranks/PEs)?	In-domain only *Ratified*	Cross-domain also	User but not common is cross-domain
HiHAT supports networking	Yes *Ratified*	No	Restricted implementation, at 1st
HiHAT supports weak memory model	Yes *Ratified*	No	Restricted implementation, at 1st
Essential/universal in core, enhancements/specialized in extension	Yes *One exception*	No *Piotr: could splinter for $*	
Support metadata for experimentation	Yes *Ratified*	No	
User is responsible for de/marshaling	Yes *Ratified*	No	
Granularity of work and communication	5us *Ratified*	10us	1us
Consider inlinable headers	Not at 1st *Ratified*	From the start	Never

SUPPLEMENTAL MATERIAL

- Features
  - Resilience
  - Static or dynamic
  - Portability, retargetability
- Explanatory
  - Services, transformations, building blocks
  - Potential interoperability conflicts
  - Hierarchy
  - Actions and resources
- Motivation
  - Motivation for dynamic scheduling
  - Motivation for uniform hetero interface
  - Motivation for task scheduling
  - Trends
  - Following the example of MPI
- Details
  - Marshaling detail
  - Granularity analysis detail

RESILIENCE

Separation of concerns between low-level infrastructure, high-level framework

- Sample goals: Know that a task failed, restart it on available resources
- Requirements of HiHAT
  - Be able to submit tasks asynchronously, get a handle back
  - Be able to kill a task
- Requirements of higher layers
  - Be able to ask for a task that experienced a fault to be killed off and cleaned up
  - Be able to associate code and data addresses that fail with handle of task that failed
  - Be able to buffer side effects and not commit them unless task is successful
  - Restarting a failed task

STATIC OR DYNAMIC

Both need a common infrastructure

- Commonalities between static and dynamic
  - Same actions: cost models, binding, ordering, allocation, data copies
  - Either can be greedy, look at a limited scope, or buffer to maximize the scope
- Similar principles, slightly different approach
  - Static vs. dynamic: make decisions, either record them for later or execute immediately
- The same (library) primitives are applicable to both
  - In order to be applicable to dynamic runtimes, can’t be *only* a compiler
  - But library interfaces need to be vetted to address compiler effectiveness and efficiency

PORTABILITY, RETARGETABILITY

- Portable: code doesn’t have to change across targets
- Retargetable: equivalent functionality is available; transformations may be applied by human tuner, or auto-tuning or automated machine-model-based heuristics
- Functional portability is achieved by expressing semantics (the “what”) cleanly
- Performance portability is achieved by abstracting the how to a target-agnostic level

→ Separate SW into

  - Above HiHAT
    - what’s not target specific, even if it’s informed by target parameters → perf portability
    - what’s responsible for functionality
  - Below HiHAT
    - what’s target specific
    - what’s responsible for target-specific performance

SERVICES

Target-agnostic pluggable services

- Build dependences
  - Convert sequence of functions into dependent tasks, or
  - Accept DAG spec
- Monitoring
  - Insert timing primitives, insert primitives that trace where & when things happen
- Visualization
  - Use enumeration to build time vs. resource matrix
  - Post-process monitoring primitive results to build event timelines
  - Show the annotated results

TRANSFORMATIONS

Pluggable operators that substitute M new actions for N old actions

- Aggregation
    - M < N, e.g. contiguous data movement, sub-sequence of tasks on same resources
- Decomposition
    - M > N, e.g. tiling, apply hierarchical refinement
- Specialization
    - Specialize the task implementation for a given memory kind or data layout
    - Manage temporary buffers: task ← moved input operands ← allocated temp buffer ← free space for move ← completed task

FUNCTIONAL BUILDING BLOCKS

Pluggable modules

- Compute costs
  - Simple: based on operand sizes, floating point arithmetic intensity factor
  - Richer: O() complexity in operand size, may depend on data layout
- Communication costs
  - Simple: based on operand size, model of bandwidth and latency for topology
  - Richer: based on data layout, e.g. contiguity, non-unit stride, whether blocked
- Scheduler
  - Simple: Earliest completion time, given data movement and compute
  - Richer: Trade off among implementations on different computing resources and with different data layouts, considering the extra costs of data re-layout

POTENTIAL INTEROPERABILITY CONFLICTS

Some challenges for interoperability

- Multiple communications libraries
  - Spec may not require interoperability or thread safety, implementations may vary
  - May hold and wait a common set of resources, leading to deadlock (portals?)
  - Unlikely to have performance isolation
  - Separating inter-rank/process/PE and intra-rank/process/PE may be insufficient
- Third-party libraries
  - Could be unexpectedly used in common by different targets

HIERARCHICAL INVOCATION EXAMPLE

- Input: sequence of function calls with operands and operand descriptors
- Root layer of hierarchy: distribute work across nodes in sub-cluster
    - Dependence analysis: discover deps among function calls; allow multiple granularities
    - Model costs: each function on each node, each data xfer between nodes
    - Convert: func → <sync on preds, move input opnds, alloc output buf, task, trigger sync>
    - Schedule: bind to nodes and preliminary order based on cost models
    - Pass down hierarchy to nodes
- Leaf layer of hierarchy: distribute work across {CPU, GPUs} resources in node
    - Configure: potentially partition resources, define # of streams
    - Model costs: each function on each {CPU, GPU}, data to/from {CPUs, GPUs}
    - Model parallelism: consider available resources and available parallelism
    - Transform: decompose appropriately, compute → <data re-layout, spcl compute>
    - Schedule: bind to {CPU, GPUs} streams, order within each stream, add alloc & sync
    - Pass sequence of {compute, data movem’t, data alloc, sync} actions to HiHAT User Layer

KINDS OF ACTIONS

Actions are the basic ingredient of asynchrony

- Compute – invoke task
- Data management – {alloc/free}x{mem kinds}, materialize, pin, marshall/demarshall
- Data movement – set up, move, fence/quiet
- Coordination – sync {wait, induce} x {data, control} x {futures} x {local, remote} x {all, any}

RESOURCES

- Logical – abstract, unbound / physical – concrete, bound
- Domains – collection of associated computation and memory resources
- Hierarchy
    - Physical domain – has shared address space
    - Coherence domain – has coherence
    - NUMA domain – has locality, could still have multiple levels, e.g. L1$, L2$, device, node
- Compute: Stream – dependences implicitly managed within, explicitly across streams
    - Resources could be semi-static or dynamically bound
    - Resources that perform enqueuing could be distinct from resources that perform execution
- Data: Buffer – collection of data elements
    - Could be pre-allocated and wrapped with metadata, or allocated on demand
    - Could have many instances, e.g. multiple shared or one that's dirty
    - Immutable properties of whole buffer: alias able
    - Mutable properties per instance: materialized, hierarchical affiliation, pinned, layout

REQUIREMENTS OF ACTIONS

- All are asynchronous and return a future
- Killable – redundant, speculative, faulty
- State is pollable
- Tracable – maybe with inferior performance
- Debuggable – not sure what this means yet

DYNAMIC SCHEDULING AND DEFERRED EXECUTION

Enable latency tolerance, performance portability

Unpredictability

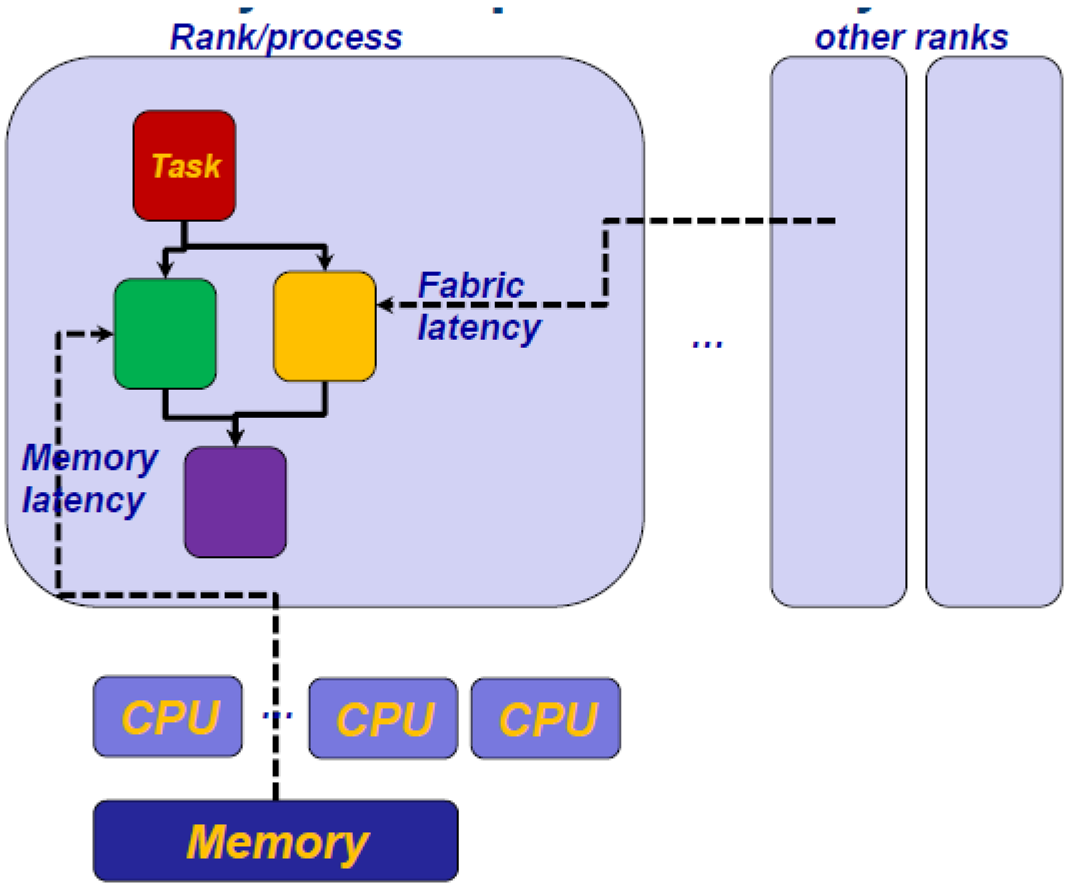
- Latency of data movement from memory a

Portability

- Capabilities for computer and fabric vary

Requires

- Asynchronous execution
- Dynamic scheduling
- Integrated dependence management
- Platform tuning



AVOID OVER-CONSTRAINING THE SOLUTION

Separation of concerns through abstraction to the rescue

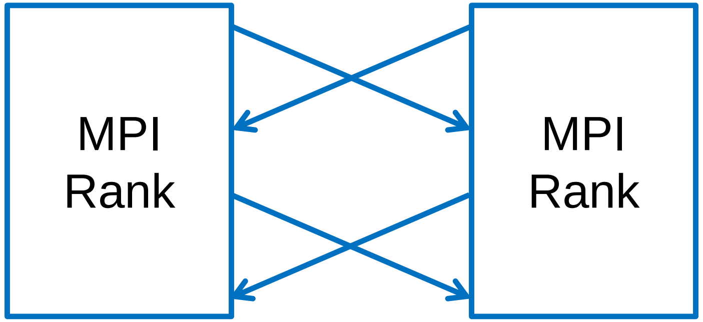
Bulk synchronous programming – the old way

- Strict phase boundaries separated by barriers
- In-order messages, all sent at the same time
- Exposure to network latency

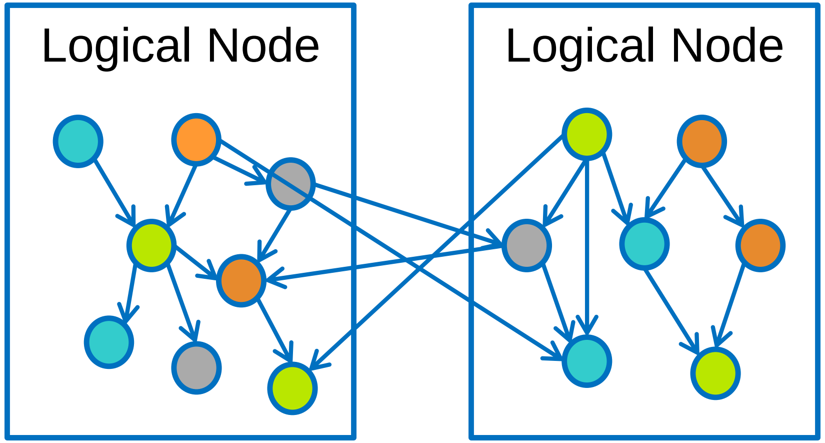
Asynchronous multi-tasking – the way forward

- Data flow execution among tasks
- Many in-flight messages, handled out of order
- Emphasis on throughput instead of latency

Messages arrive in order



Messages arrive in any order



DYNAMIC SCHEDULING OR PURELY STATIC SCHEDULE

Let the runtime help manage unpredictability

Problem

- Increasingly difficult to create a portable and effective static schedule, given variances and uncertainties from DVFS, network congestion, multiple memory kinds, unpredictable load imbalance, and the need for fault tolerance

Solution

- Statically schedule where you can, dynamically schedule tasks and data movement where needed. Data movement among MPI ranks and to/from various memory kinds is all integrated into a unified framework
- Locality the scope of dependence analysis and scheduling to promote efficiency
- Open-sources code with pluggable interfaces enables customization
- An optional hierarchy for the scope of binding and scheduling, that can help facilitate localization and fault tolerance

UNIFORM HETERO INTERFACE OR COLLECTION OF INTERFACES

Let the platform ninjas do the hard work

Problem

- Disjoint interfaces for each target can be confusing and inefficient
- Best allocators may vary by kind and over time

Solution

- Offer a uniform interface for all computing targets
- Offer a unified, declarative interface for all memory kinds to complement imperative interfaces

PARALLEL TASKS REQUIRES… TASKS

Dusty-deck Fortran loop nests become a sequence of task invocations

```
!  Fragment 1 z=f(x,y)
{ loopA
     loopB
     loopC

!  Fragment 2 u=f(v,z)
{ loopD
     loopE

!  Fragment 3 w=f(t)
{ loopF
     loopG
```

```
function frag1(x, y) result(z)
loopA
    loopB
    loopC
end function frag1
function frag2(v, z) result(u)
 loopD
    loopE
end function frag2
function frag3(t) result(w)
 loopF
    loopG
end function frag3
```

```
program xfunc
z=frag1(x,y)
u=frag2(v,z)
w=frag3(t)
end program
```

TASKS + LOOPS OR JUST ONE OR THE OTHER

Seamlessly support task and loop-level parallelism

Problem

- Modern platforms may support a high degree of heterogeneous parallelism
- Task parallelism and thread parallelism within tasks may both be moderate
- Computing and memory components may be distributed enough that data movement latencies may become a bottleneck

Solution

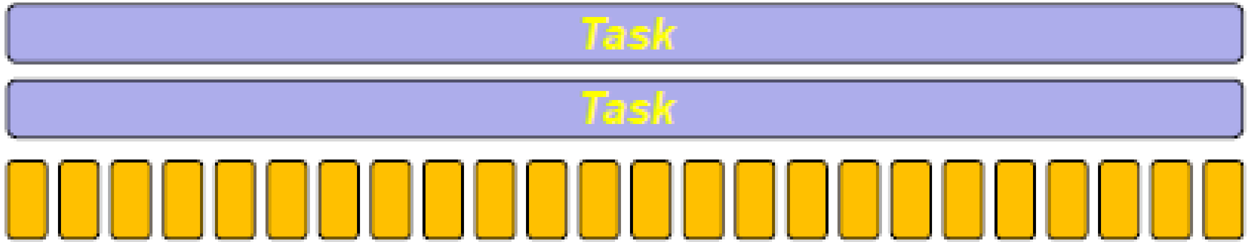
- Support task parallelism across nodes and within computing elements of nodes
- Enable overlapping of communication and computation
- Platform-tuned primitives offer best performance

Comparison

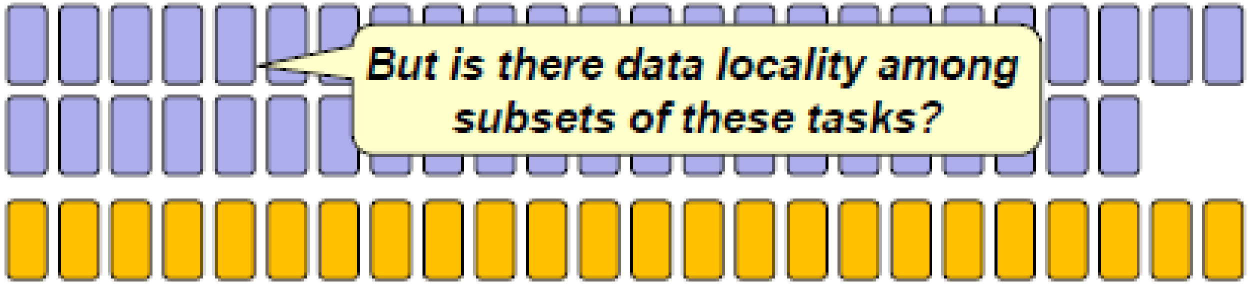
- OpenMP: while task parallelism within a node is possible, it requires expertise
- OpenMP enables async, but doesn’t yet support offloading to multiple node types

TASK SCHEDULING IS TRIVIAL IF

- Each task is so parallel it fills a whole resource

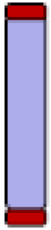


- There are so many tasks that each can execute on one thread

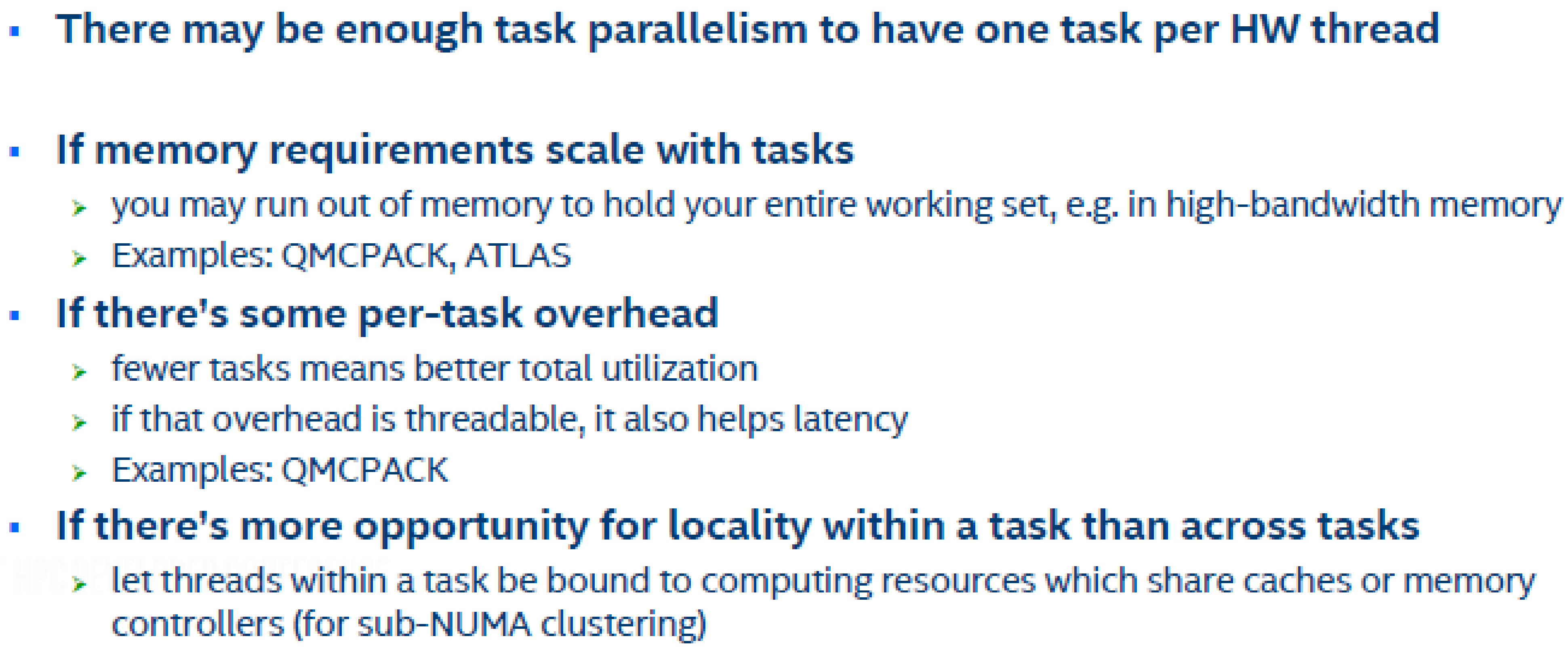


But is there data locality among subsets of these tasks?

- Tasks execution drowns out communication latency



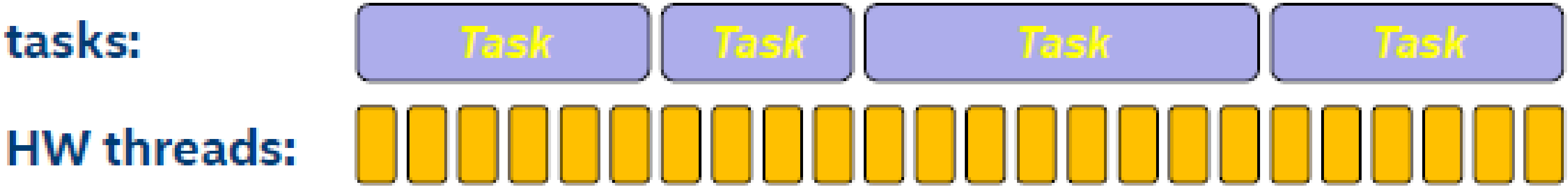
ADDITIONAL REASONS TO USE FEWER TASKS



- **There may be enough task parallelism to have one task per HW thread**
- **If memory requirements scale with tasks**
  - you may run out of memory to hold your entire working set, e.g. in high-bandwidth memory
  - Examples: QMCPACK, ATLAS
- **If there's some per-task overhead**
  - fewer tasks means better total utilization
  - if that overhead is threadable, it also helps latency
  - Examples: QMCPACK
- **If there's more opportunity for locality within a task than across tasks**
  - let threads within a task be bound to computing resources which share caches or memory controllers (for sub-NUMA clustering)

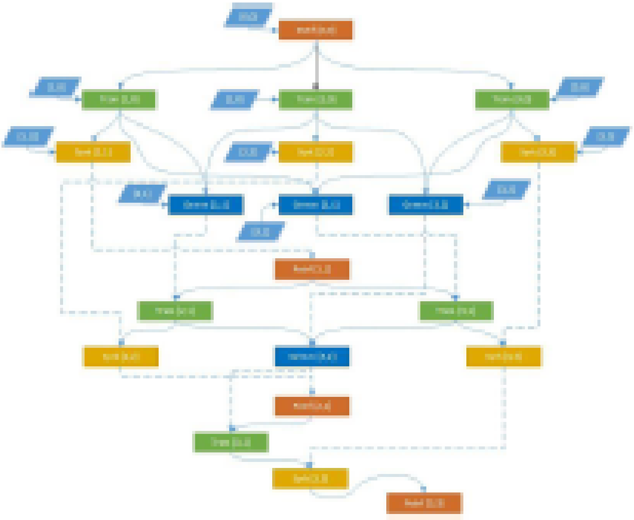
TASK SCHEDULING IS MORE INTERESTING WHEN

- There are several concurrent tasks, but they have limited parallelism

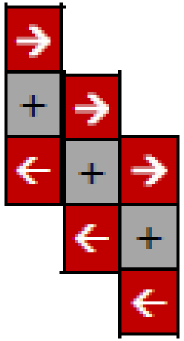


See backup for data on efficiency vs. size and width

- Tasks have dependences among them, but the latency of execution and/or communication is unpredictable

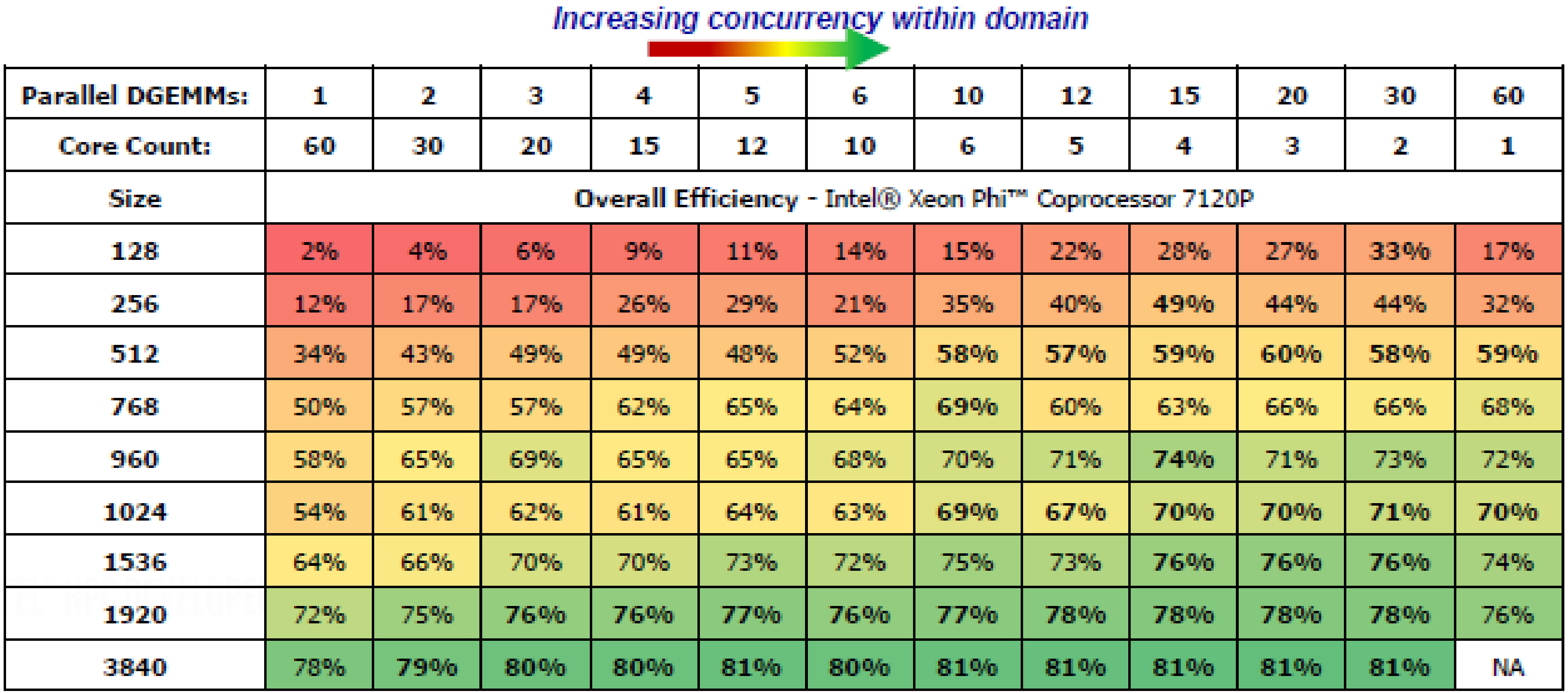


- It becomes important to pipeline communication and computation



- The target platform is heterogeneous

PARALLEL SMALL TASKS IMPROVE EFFICIENCY



Increasing concurrency within domain

Parallel DGEMMs:	1	2	3	4	5	6	10	12	15	20	30	60
Core Count:	60	30	20	15	12	10	6	5	4	3	2	1
Size	Overall Efficiency - Intel® Xeon Phi™ Coprocessor 7120P											
128	2%	4%	6%	9%	11%	14%	15%	22%	28%	27%	33%	17%
256	12%	17%	17%	26%	29%	21%	35%	40%	49%	44%	44%	32%
512	34%	43%	49%	49%	48%	52%	58%	57%	59%	60%	58%	59%
768	50%	57%	57%	62%	65%	64%	69%	60%	63%	66%	66%	68%
960	58%	65%	69%	65%	65%	68%	70%	71%	74%	71%	73%	72%
1024	54%	61%	62%	61%	64%	63%	69%	67%	70%	70%	71%	70%
1536	64%	66%	70%	70%	73%	72%	75%	73%	76%	76%	76%	74%
1920	72%	75%	76%	76%	77%	76%	77%	78%	78%	78%	78%	76%
3840	78%	79%	80%	80%	81%	80%	81%	81%	81%	81%	81%	NA

Credit for data collection: Efe Guney

TRENDS

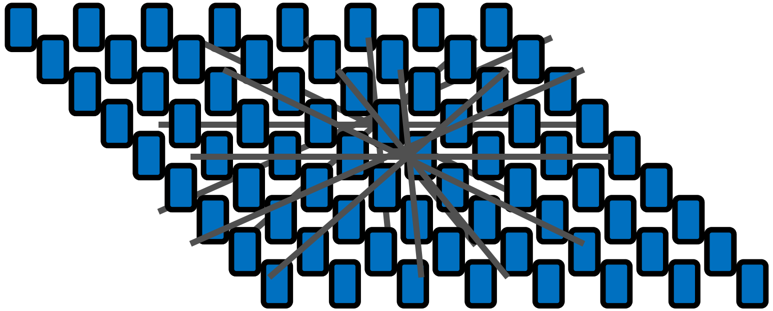
Relevant to small or large scale HPC, AI

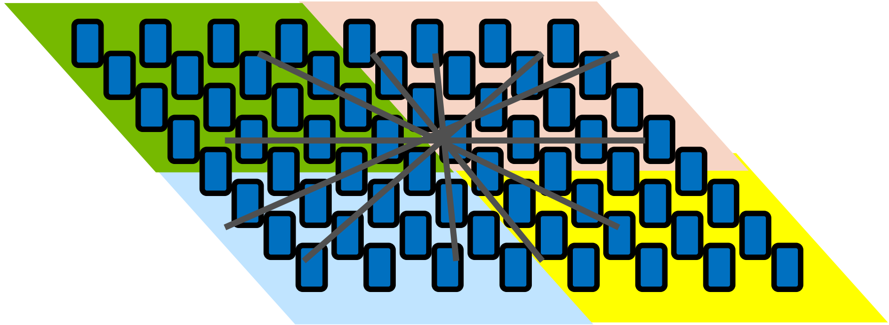
- Scale → **Hi**erarchical
- Differentiation for efficiency → **H**eterogeneity
- Unpredictability → **A**synchronous
- Functional and data parallelism → **T**asking

TRENDS

Relevant to small or large scale HPC, AI

- Scale → Hierarchical
  - Locality: higher effective bandwidth, lower latency, better TLBs
  - Abstractions that are repeatable at various levels and granularities

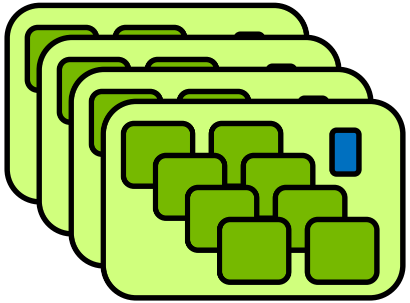




TRENDS

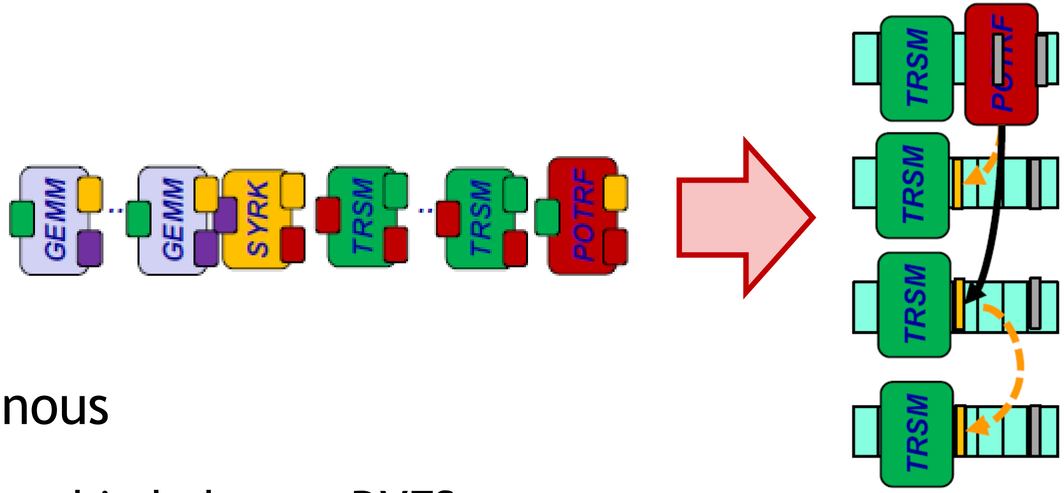
Relevant to small or large scale HPC, AI

- Differentiation for efficiency → Heterogeneity
  - Throughput and latency cores
  - Power efficiency
  - Higher aggregate bandwidth



TRENDS

Relevant to small or large scale HPC, AI



- Unpredictability → Asynchronous
  - Varied progress: dynamic load imbalance, DVFS
  - Network congestion
  - Depth in memory hierarchy
  - ➢ *Bind and order actions from a queue onto resources with dynamic scheduling*

TRENDS

Relevant to small or large scale HPC, AI

- Functional and data parallelism → Tasking
  - Enqueue(Name, <Operands>, <Optional descriptors>)
  - Transformations: decompose, aggregate, substitute

TRENDS

Relevant to small or large scale HPC, AI

- Scale → **Hi**erarchical
- Differentiation for efficiency → **H**eterogeneity
- Unpredictability → **A**synchronous
- Functional and data parallelism → **T**asking

HiHAT

GOALS, FROM SECTION 1.1 OF MPI SPEC

Inspired by a success story

- Fundamental to the environment
- API: library, not a language
- Heterogeneous environment: portable, easy to use
- Retargetable to many vendor platforms: clear and common interface
- Convenient C and Fortran bindings, language-independent semantics
- Part of the soul of MPI, also relevant to HiHAT
- Efficient communication: enable distributed systems
- Reliable communications interface
- Thread safe

Feedback item

USER FUNCTION ARGUMENT MARSHALING

Closure 101

Operationally, a closure is a record storing a function together with an environment: a mapping associating each free variable of the function (variables that are used locally, but defined in an enclosing scope) with the value or reference to which the name was bound when the closure was created.[1]

A free variable is a notation that specifies places in an expression where substitution may take place[1].

Software constructs: callbacks, lambda functions, nested functions, contained functions(?), function objects

Similar implementation constructs: task, OpenACC/OpenMP “kernels” – with marshalling

[1] wikipedia

Feedback item

USER FUNCTION ARGUMENT MARSHALING

OpenMP task implementation case study

```
#pragma omp task default(firstprivate) shared(a,b,c)
{ … }
```

User responsible for ensure a,b,c remain in “scope” until task completes.

```
void ompTask_foo$1(void *args)
{ <copyout firstprivate values>;
  ptr_a = args[a_location];
  ptr_b = args[b_location];
  ptr_c = args[c_location];
… }

foo$1_args = {firstprivate values, shared object addresses}

Task_struct task1 = {ompTask_foo$1, foo$1_args};

enqueue(&task1);
```

Feedback item

USER FUNCTION ARGUMENT MARSHALING

OpenMP target implementation case study

```
#pragma omp target map(a,b,c)
{ … }
```

User responsible for ensure a,b,c remain in “scope” until target completes.

```
map_a_addr = omp_target_alloc(sizeof(a), <device id>); //using user exposed functions for compiler generated work
map_b_addr = omp_target_alloc(sizeof(b), <device id>);
map_c_addr = omp_target_alloc(sizeof(c), <device id>);
omp_target_memcpy( map_a_addr, &a, sizeof(a), 0, 0, <device id>, <host id>);
…
omp_target_associate_ptr(&a, map_a_addr, sizeof(a), 0, <device id>);
…
foo$1_args = {firstprivate values, map_a_addr, map_b_addr, map_c_addr};
target_struct target1 = {ompTarget_foo$1, foo$1_args,<device id>};
target_enqueue(&task1);
<copy out>
<disassociate>
<free>
```

Feedback item

USER FUNCTION ARGUMENT MARSHALING

Case study summary

Avoided OpenCL because it is extremely lowlevel, however, it provides its own data marshalling tools

Compilers use marshalling to implement both OpenMP and OpenACC directives

Data marshalling does not mean closures are provided since a closure is both a functor and data.

OpenMP and OpenACC both marshall directly to device memory

> *Why? “performance”*
>
> *What is wrong with this? “computation must follow data if memory is not ‘shared’”*

Feedback item

USER FUNCTION ARGUMENT MARSHALING

Memory marshalling

Marshall direct to device memory?

- Faster launch of work once resource becomes available
- Allocation: size and device as input, void* as output
- Data transfers source, destination, size, device id if memory addresses do not care info

Marshall via a buffer?

- Launch slower because it must move “buffers”
- Launch not tied to a single device
- Allocation, size as input, identifier as output
- Data Transfer, source pointer buffer id and size as inputs

No need for types only sources “addresses”,

Feedback item

USER FUNCTION ARGUMENT MARSHALING

Task marshalling

Function pointer

- Needs to be something the underlying support libraries can invoke
- How does programmer get value?
- How does runtime find callee?

Arguments

- Kernel arguments can be free or nearly so for some architectures
- Variadic functions not the nicest thing to program to.
- Push a blob of data or gather blobs as kernel arguments
  - As single blob is easer for the runtime

Function pointer plus argument blob => closure

Feedback item

USER FUNCTION ARGUMENT MARSHALING

Summary

Closure – good way to think about tasks and “off-loading”

User is responsible for marshaling and de-marshaling

- *Compiler could do marshalling but that means compiler involvement!*

Avoid considerations of operand size, type (type casting)

- *As seen on previous slides if caller handles sizes and types than runtime only needs a small number of interfaces and no need to handle arbitrary user types*

Existing accelerator programming models provide data marshalling APIs only

GRANULARITY ASSUMPTIONS

Hi-level Job Flow cost – open for debate

Marshall data (0us to 100ms or more) User responsible for optimizing this

Build data + function closure – get the task ready – (4 – 100 cycles per item)

Setup dependency info for closure – ensure new task will run at the right time – (STA)

Enqueue closure – hand task over to the runtime for dispatch – (11 cycles)

Resolve dependencies – move tasks from waiting to runnable state – (0us to ?)

Dispatch closure

- Data movement – move the data section of the closure to the device – (4 cycles ?)
- Call closure function on data – (1us to 10us)

Return status – (1 cycle)

GRANULARITY ASSUMPTIONS

Assume data marshalling is optimized by client so cost approaches zero

Assume 5 control words, found in L, plus function pointer in closure

Assume no dependencies—fastest path

Estimated cost of launch

- 24 cycles to build closure
- 11 cycles to enqueue closure
- 10us to start working – assuming offloading to device other than enqueuing device

Any kernel running for less than 10us will spend more time getting ready to run than running on some machines

GRANULARITY ASSUMPTIONS

inlinable header or lib-embedded function call

What specializations would an inlinable header help with?

- *Device type, device selection*

How can they co-exist?

- *Of course the library just has to provide more APIs to short circuit some control flow*
- *Is there sufficient gain for the maintenance cost?*

OpenACC has device_type(type) clause

- *Allows directive to be tailored for a given target, important when optimizing something like gangs, workers, and vector length.*
- *Allows a single compile to target multiple devices "optimally"*
- *OpenMP is talking about similar idea*

Expected savings likely in the order of 100's of cycles at best, well below the 10us of launch overhead assumed on previous slide!