Status of LLNL Monte Carlo Transport Codes on Sierra GPUs

M&C 2019

M. Scott McKinley, Ryan Bleile, Patrick Brantley, Shawn Dawson,
Matt O'Brien, Mike Pozulp, David Richards

August 28, 2019

LLNL-PRES-787821

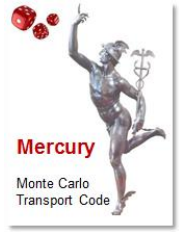
This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC

Lawrence Livermore National Laboratory

Overview

- Mercury and Imp share common code to help with porting
- Porting to the GPU relies on new nuclear data format and collision physics library
- Porting issues
  - Memory model
  - Fat / thin threads
  - Tally replication
  - History vs event based algorithm
  - Hybrid CPU+GPU load balancing
  - Nuclear data routines on the CPU+GPU
- Results
- Future work

Mercury is the next-generation general purpose Monte Carlo particle transport code under development at LLNL



- Transports neutrons, photons, and light element (hydrogen and helium) charged particles
  - Transport is super thermal with special thermalization models for neutrons
- Relativistic physics
- Treats fixed source and criticality problems
- Parallelized via domain replication and domain decomposition
- Written in C++ with a Python user interface
- Runs efficiently on current generation massively parallel computing platforms

Imp implements the standard implicit Monte Carlo multigroup algorithm for thermal x-ray photon transport for the frequency-dependent case

- The frequency-dependent photon transport equation in the absence of scattering and external sources is given by*

$$\frac{1}{c}\frac{\partial I\left(\underline{r},\underline{\Omega},\nu,t\right)}{\partial t}+\underline{\Omega}\cdot\underline{\nabla}I\left(\underline{r},\underline{\Omega},\nu,t\right)+\sigma_a\left(\underline{r},\nu,T\right)I\left(\underline{r},\underline{\Omega},\nu,t\right)=\sigma_a\left(\underline{r},\nu,T\right)B\left(\nu,T\right)$$

- The Planck function is given by

$$B\left(\nu,T\right)=\frac{2h\nu^3}{c^2}\frac{1}{exp\left(\frac{h\nu}{kT}\right)-1}$$

- The transport equation is coupled to a material energy balance equation given by

$$\frac{\partial E_m\left(\underline{r},t\right)}{\partial t}=\rho\left(\underline{r},t\right)c_v\left(\underline{r},T\right)\frac{\partial T\left(\underline{r},t\right)}{\partial t}=\int_0^\infty d\nu'\int_{4\pi}d\underline{\Omega}'\sigma_a\left(\underline{r},\nu',T\right)I\left(\underline{r},\underline{\Omega}',\nu',t\right)$$
$$-\int_0^\infty d\nu'\int_{4\pi}d\underline{\Omega}'\sigma_a\left(\underline{r},\nu',T\right)B\left(\nu',T\right)\quad,$$

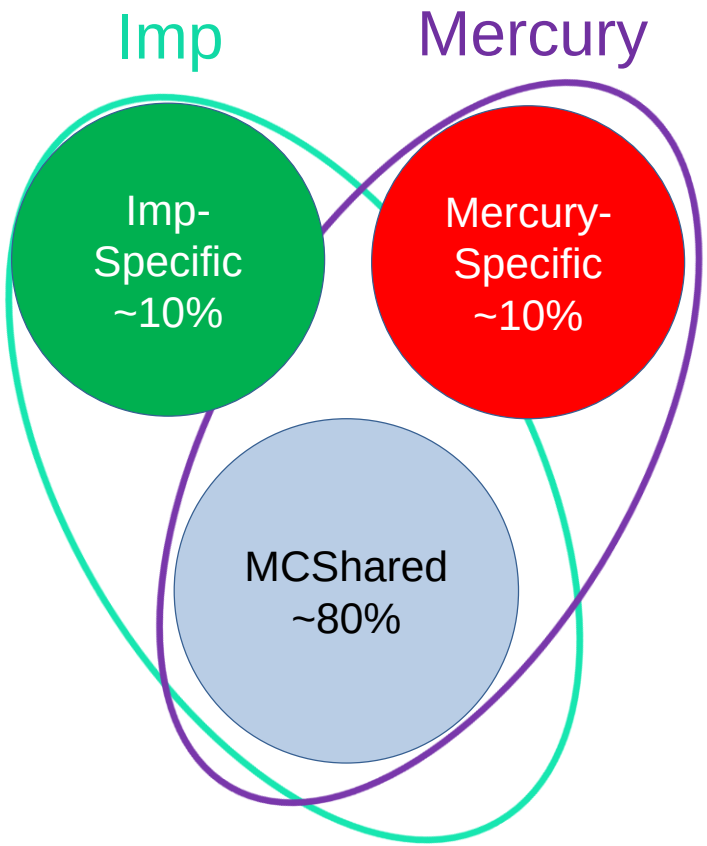
- Source tilt and random walk acceleration algorithms implemented

The development of Imp has been strongly influenced by and benefited from the algorithmic implementations in and experience gained from the Kull IMC photon transport capability

* Pomraning (1973)

The Imp IMC thermal photon transport capability is being developed using Monte Carlo infrastructure shared with the Mercury Monte Carlo particle transport capability

- MCShared: shared Monte Carlo infrastructure
  - geometry via constructive solid geometry or mesh
  - particle tracking through geometry
  - MPI/OpenMP parallelism and load balancing
  - standard and user-defined tallies
  - variance reduction
  - random number generators
  - general sources
  - input parsing, restart/checkpoint, visualization/graphics files
  - edit and plotting capabilities, etc.
- Imp: IMC-specific parts of the source code base
  - effective scattering, thermal emission, opacities, scattering physics, source tilting, random walk acceleration, etc.
- Mercury: particle transport-specific parts of the source code base
  - nuclear data and collision physics functionality, particle-specific collision functionality, criticality eigenvalue solvers, etc.
- Imp-specific or Mercury-specific code capabilities are combined with MCShared to produce either Imp or Mercury
- MCShared approximately 80% of the source code base, with Mercury- and Imp-specific parts of the source code base comprising ~10% each



A large fraction of the consolidated source code base is reused between Imp and Mercury

The porting of Mercury to the Sierra GPUs relies on the Generalized Nuclear Data Structure (GNDS)/Generalized Interaction Data Interface (GIDI) nuclear data infrastructure

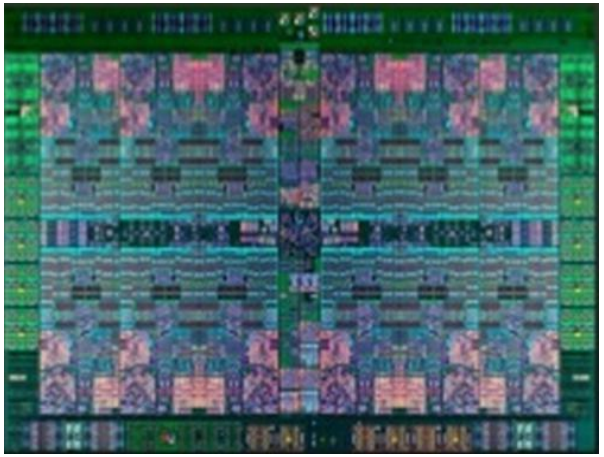
- Working Party on Evaluation Cooperation (WPEC) formed a new sub-group (SG38) to create GNDS
  - Titled “Beyond the ENDF format: a modern nuclear database structure”
- GNDS
  - Is hierarchical in a format like XML or HDF5
  - Supports any projectile/target
  - Works with an external database to give consistent nuclear masses, levels and lifetimes
  - Stores units and interpolations with data
  - Allows for experimental, evaluated and processed data to be stored in the same format/location

Generalized Interaction Data Interface (GIDI)

- Interface to accessing data in GNDS format
- MCGIDI is the Monte Carlo component that samples reactions
  - This is the only component currently ported to the GPU
- Benefits of GNDS/GIDI over legacy solution
  - More accurate sampling of the data since it can use the pdf and functional forms of the data
  - Versatility of temperature and energy bin boundary data due to processing data on the fly*
  - Photonuclear reactions
  - Continuous energy, fixed grid* and multigroup available for most trackable particles
  - Unresolved resonance treatment as pdf instead of a few lines
  - Heating* of Thermal Scatter Law (a.k.a. S(α,β)) molecular data
  - Continuous energy gain*, energy deposition and energy production cross sections
    - Also supports more accurate heating of energy deposition data
  - Support new projectile types*
  - Will allow for better data sharing via GNDS between the labs for code comparisons

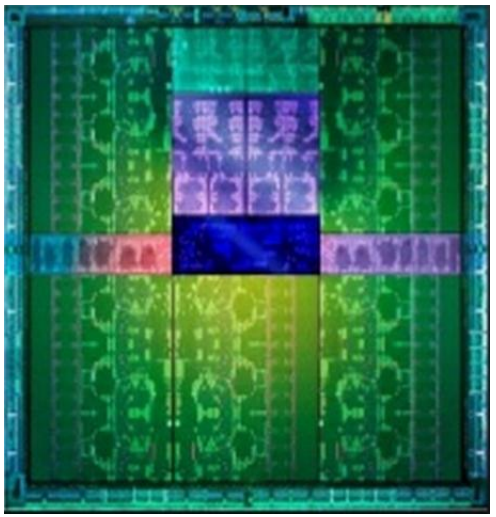
* Not yet implemented

Next generation architecture is GPU centric



IBM Power 9 CPU
<10% Sierra Flops

~4200 nodes
Each node has:
40 Power9 CPU Cores
4 Volta GPU

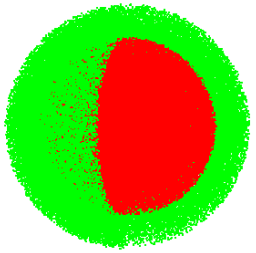


Nvidia Volta GPU
> 90% Sierra Flops
16 GB HBM/GPU

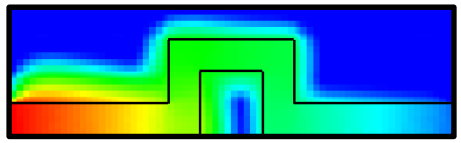
The Mercury particle transport and Imp IMC thermal photon transport capabilities have been initially ported to the Sierra GPU architecture

- Sierra GPU port of both capabilities facilitated by the evolution of the particle and thermal photon transport capabilities into a consolidated source code base
  - Separate Mercury particle transport and new Imp IMC thermal photon transport capabilities are now built from shared infrastructural source code
- Initial GPU porting approach:
  - "big kernel" history-based particle tracking kernel based on early research
  - Cuda managed memory
- Mercury and Imp physics results on the GPU are correct
  - Nightly test suite compares GPU to CPU results

Mercury Calculation



Imp Calculation



- Current Monte Carlo simulation results exhibit some slowdowns and some speedups on the Sierra GPUs compared to a CTS-1 node
  - Hybrid CPU+GPU approach improves performance overall
  - Imp IMC photon transport generally exhibiting better performance than Mercury particle transport
- Monte Carlo history-based transport is not amenable to typical GPU fine-grained threading
  - Particle tracking loop is thousands of lines of branchy, latency sensitive code
- Current focus is on refactoring to improve GPU performance, investigating CPU+GPU hybrid approach, and beginning to research event-based/kernel splitting approaches

We have chosen to initially use managed memory to facilitate porting to Sierra

- Memory Allocation
  - Memory allocation on the GPU is very slow – best to avoid
- Copying Memory
  - Too much data to easily copy by hand
  - Need to make sure the calculation pays for the overhead of copying memory
  - CudaMallocManaged provides universal addressing from CPU and GPU
    - About 300 times slower than malloc
    - Either page (4k) or texture (0.5k) minimum allocation
      - Easy to run out of memory
    - Mitigate by allocating large memory chunk and using Umpire, a third party memory pool library
- C++ classes
  - CudaMallocManaged can support pointers
    - It can not handle pointers to functions
    - Therefore, it cannot handle virtual functions in a class
    - Have to build objects on the GPU that have virtual functions
  - Does not support the Standard Template Library (STL)
    - Mixed success with Thrust
- Code does a lot of integer arithmetic, memory lookups and Boolean decisions
  - Low number of floating point operations

We implemented a new thin threading model to facilitate porting to the Sierra GPUs

Fat thread

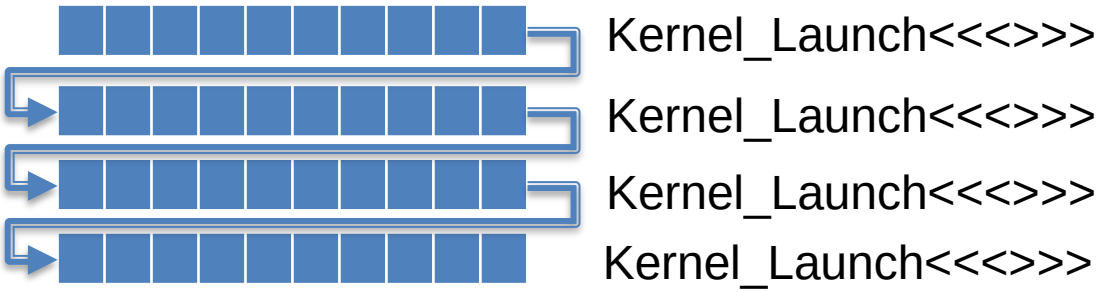
- Dozens of active threads
- Separate container of particles for each thread
- Data races managed with replication
- MPI tightly integrated in tracking loop
- Works well on CPUs

Thin thread

- Thousands of active threads
- All threads share a common container of particles
- Data races managed with atomics
- No MPI in tracking loop
- Works on GPUs and CPUs

Thin Thread Particle Tracking Loop

- Our Particle_Vault data structure is an array of pointers to Chunks of particles
- We execute one kernel launch per chunk of particles
- Before each kernel launch, we make sure we have enough pre-allocated extra chunks for created particles
- When en-queueing created particles, we atomically get a slot index to store the particle
- We do MPI streaming communication between kernel launches



We implemented variable replication of data structures to reduce memory and race conditions

- Two main types of tallies
  - Low memory but accessed a lot such as the balance tally
    - Counts number of collisions, number of fissions, etc.
  - High memory ones like scalar flux
    - Array over particle type, cell, and energy
- Under our previous fat thread mode, each tally was replicated over threads and MPI ranks
  - With thousands of GPU threads, this will no longer work due to memory constraints
- Atomics on tallies can create bottlenecks with certain types of tallies like the balance tally
- Our solution was for each type of tally to have its own independent replication level
  - Full replication is no longer required since atomics will still be used, but conflicts may be rarer

Monte Carlo History / Event Based Algorithm

- History Based Algorithm
  - For each particle, computes the distances to each possible event (i.e. Census, Facet Crossing, Collision, etc.)
    - Low divergence
  - Perform Event
    - High divergence
  - Repeat
- Event Based Algorithm
  - Computes the distances to each possible event for all particles
    - Low divergence
  - Sort particles into queues of similar events
    - Could be slow or have small pools
  - Perform the event on the pools of particles
    - Low divergence
- Speculation was event modeling could pay off well on a GPU device
- Testing from a small mini-app showed the performance of history- and event-based were similar on the GPU
- Results from ORNL showed some good gains from event based
- We expect event based approach to lead to lower register pressure with more kernel calls

In a problem in which MPI ranks control a CPU or GPU processor(s), how do you load balance particles?

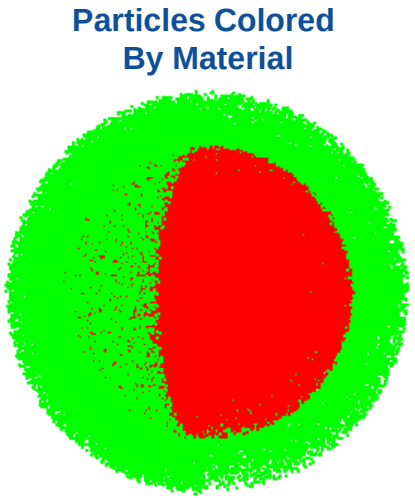
- Each rank could control 1 or more processors
  - e.g. A rank on CPU could be threaded
- Calculate the average speed in particles per second of MPI ranks with GPUs and MPI ranks without GPUs
  - Speed is determined based on previous cycle
- To balance the workload, ranks with too many particles send them to ranks with too few so that every rank is expected to run the same amount of time
  - Using a min/max priority queue
- Using this resulted in 1.8X speedup in Mercury and 4.3X in Imp for one particular problem

GPU Porting Progress of MCGIDI

- MCGIDI for Monte Carlo is written in C++ and uses polymorphism with virtual functions
  - Managed memory does not work with virtual functions
- Wrote a serialize function to pack or unpack the data into a data stream
  - Read in on CPU; pack data; copy to GPU; unpack on GPU
  - Unpacking involves placement new operations
- Changed exception handling from throws to print statements
- Added classes to mock up std::string and std::vector
- Deep call stack is not optimal for GPUs

We assessed Mercury neutron transport on Sierra using a Godiva in water test problem

- Godiva critical sphere surrounded by water*, calculate α eigenvalue
- Combinatorial geometry, continuous energy nuclear data, 4×10^6 MC neutrons per time step
- 36 CTS-1 cores, 40 P9 CPU cores, 4 V100 GPUs, or 4 V100 GPUs + 36 P9 CPU cores
- Mercury GPU physics results in excellent agreement with CPU results
  - α = 9.50437 μs^{-1} agrees to 5 digits, well within statistics
- 4 GPU+36 core simulation exhibits a ~1.2X overall speedup compared to a CTS-1 node
  - Particle tracking ~1.5X faster on 4 GPUs+36 cores

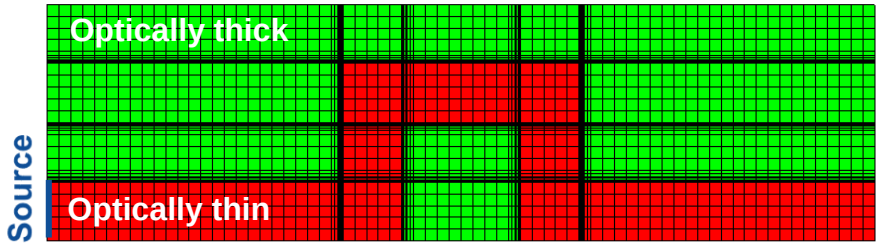


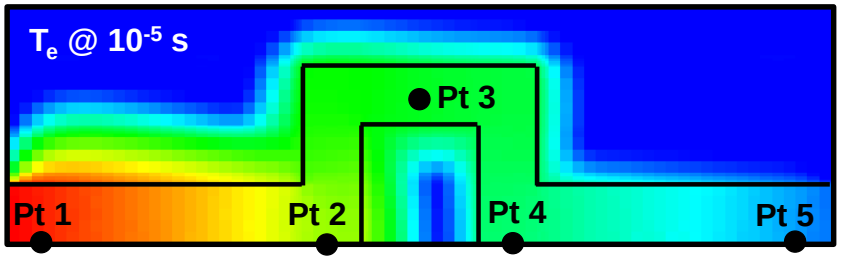
Resources	CPU / GPU	Total Time [minutes]	Particle Time [minutes]	Init/Final Time [minutes]
CTS-1	36 cores	1.43	1.28	0.16
P9	40 cores	1.76 (0.82X)	1.54 (0.83X)	0.22 (0.72X)
V100	4 GPUs	1.76 (0.81X)	1.59 (0.81X)	0.18 (0.89X)
V100+P9	4 GPUs + 36 cores	1.19 (1.20X)	0.88 (1.45X)	0.31 (0.50X)

* D. E. Cullen, C. J. Clouse, R. Procassini, R. C. Little, "Static and Dynamic Criticality: Are They Different," UCRL-TR-201506 (2003)

We assessed Imp IMC thermal photon transport on Sierra using the Crooked Pipe test problem*

- Idealized radiation transport test problem*
  - RZ geometry, constant gray opacity, constant specific heat, 300 eV black body source
- 20×10^6 MC photons per time step, simulation run to 10^{-5} s with variable time step (1,063 cycles)
- 36 CTS-1 cores, 40 P9 CPU cores, 4 V100 GPUs , or 4 V100 GPUs + 36 P9 CPU cores
- Imp IMC GPU+CPU physics results in excellent agreement with CPU results and with Kull IMC
- 4 GPU+36 core simulation exhibits a ~2.4X overall speedup compared to a CTS-1 node
  - Particle tracking ~2.5X faster on 4 GPUs+36 cores





* F. Graziani, J. LeBlanc, "The Crooked Pipe Test Problem," UCRL-MI-143393 (2000)

Resources	CPU / GPU	Total Time [minutes]	Particle Time [minutes]	Init/Final Time [minutes]
CTS-1 [run136]	36 cores	471.4	458.8	12.67
P9 [run137]	40 cores	561.6 (0.84X)	524.6 (0.87X)	36.95 (0.34X)
V100 [run138]	4 GPUs	236.7 (1.99X)	226.2 (2.03X)	10.51 (1.21X)
V100+P9 [run139]	4 GPUs + 36 cores	193.4 (2.44X)	181.9 (2.52X)	11.56 (1.10X)

Conclusions

- We have developed the new Imp IMC capability based on infrastructure shared with the Mercury Monte Carlo code
- Both Mercury and Imp have been initially ported to run on the Sierra GPU architecture
- The physics results of both capabilities are correct, and we are working to improve the GPU performance
- The development of consolidated capabilities enables us to tackle the ongoing challenges posed by the Sierra GPU architecture for Monte Carlo particle and photon transport in a consistent framework

Future Strategy

- Continue performance optimization
- Investigate event-based and kernel splitting approaches for tracking particles on the Sierra GPU architecture
- Look at Unity builds to see if call stack sizes can be decreased
  - Start with MCGIDI and then see if the whole code can be built this way
- Look at ways to simplify the collision event routines so Mercury's collision speed can get closer to Imp's speed

Backup slides

The LLNL Monte Carlo Transport Project is developing a new IMC thermal photon transport capability called Imp using infrastructural components shared with the Mercury Monte Carlo particle transport code

- Imp implements the standard implicit Monte Carlo multigroup algorithm for thermal x-ray photon transport for the frequency-dependent case

- A major driver for this code sharing approach is to enable a consolidated source code base for porting to the LLNL Sierra supercomputer
  - Sierra: IBM/Nvidia machine with two IBM Power9 CPUs (44 cores) and four Nvidia Volta V100 GPUs per node

- This shared source code base enables:
  - advanced architecture implementations to benefit both Mercury and Imp
  - common use of code infrastructure/capabilities: geometry representation, particle tracking through geometry, MPI/OpenMP parallelism, load balancing, standard/user-defined tallies, variance reduction, input/output for parsing & restart/graphics files, etc.
  - project staff to have expertise in both implementations for debugging/user support

An Implicit Monte Carlo Scheme for Calculating Time and Frequency Dependent Nonlinear Radiation Transport*

J. A. Fleck, Jr. and J. D. Cummings

Lawrence Radiation Laboratory, University of California, Livermore

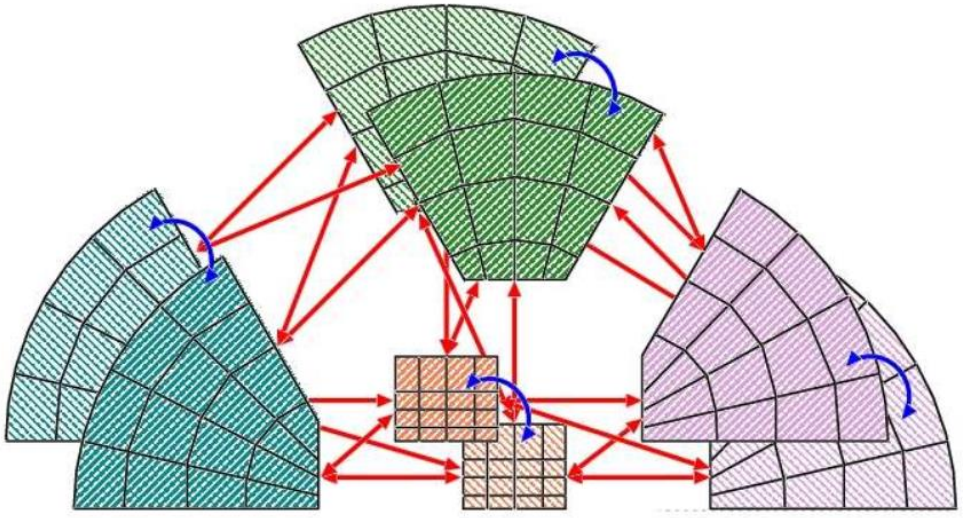
Received January 13, 1971

Mercury Parallelization Model

- Supervisor MPI Ranks
  - In control of a spatial section of the geometry
- Worker MPI Ranks
  - Replicate the work of a supervisor
  - May be moved around as needed for load balancing
- A MPI rank may thread its work if additional computational resources are available

Particle Communication Between Adjacent Spatial Domains

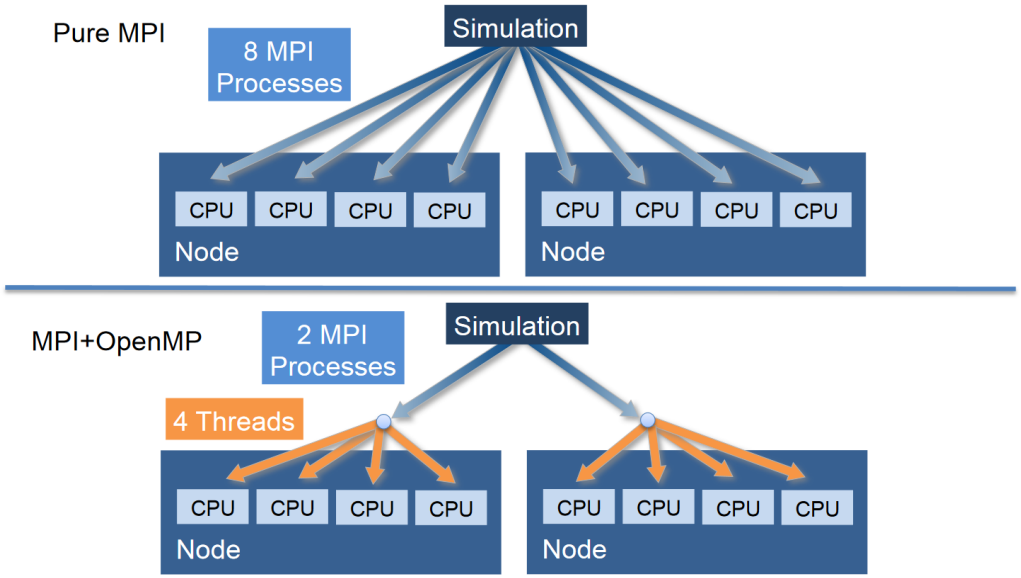
"Summing" Communication Between Replicated Spatial Domains



Domain Decomposition and Domain Replication
(4-way Spatial and 2-way Particle Parallelism)

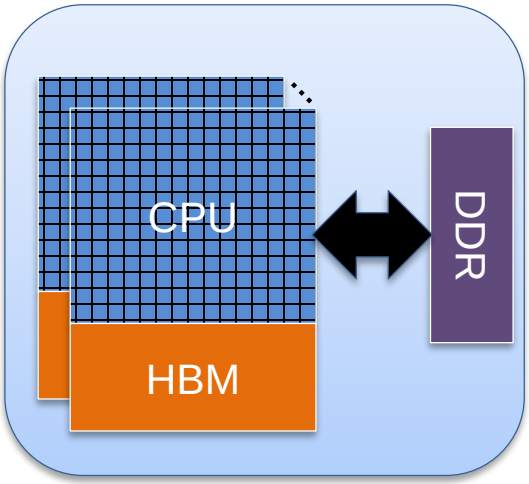
From MPI to MPI+OpenMP

- Until relatively recently, MPI has been the main mechanism of porting to new computer platforms
  - OpenMP threading was available, but it was always slower
  - Scaled to millions of processors using both MPI and OpenMP
- In more recent hardware, OpenMP threading became fast enough and could save memory
  - In many problems it sped up the initialization and finalization sections of a cycle due to threaded loops over zones
- Shared memory MPI has been used for the nuclear data



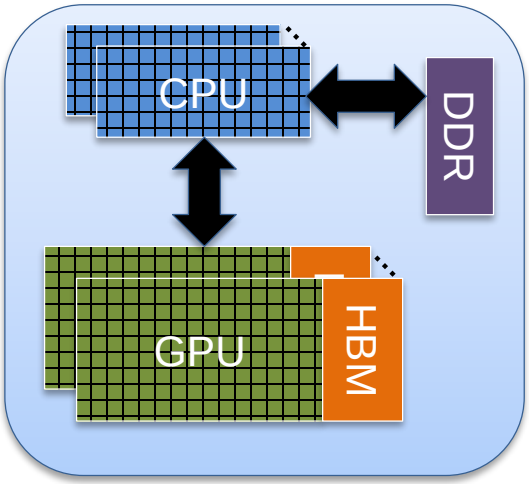
Advanced Architectures

- Many-Core (like Trinity)
  - Many relatively small homogeneous compute cores
  - 10's of thousands of nodes with millions of cores
  - Multiple levels of memory
  - Single/Dual rail high performance network



Notional Many-Core Node

- Hybrid Multi-Core (like Sierra)
  - Multiple CPUs and accelerators per node
  - Small(ish) number of very powerful nodes
  - Multiple levels of memory
  - Multi-rail high performance network



Notional Hybrid Multi-Core Node

GPU Challenge - Divergence

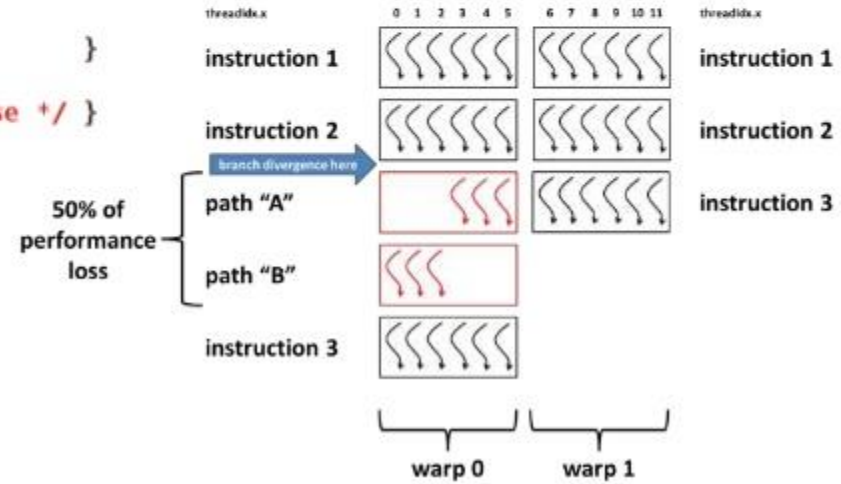
Graphic Processor Unit

Branch Divergence

```
// instruction 1
// do something

// instruction 2
if(threadIdx.x > N)
{ /* do something */      }
else
{ /* do something else */ }

// instruction 3
// do something
```



10 / 17

Current Code Flow