Sandia National Laboratories

SAND2020-11993PE

Achieving Performance on Next-Generation Hardware with PIC Codes

EMPIRE

Presented By

Matthew Bettencourt

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & 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.

Outline

Walk down memory lane
- What is ”next generation” mean
- Why do I care – my argument is that you should or will

What are the hard parts of ”next generation” computing
- We have to reconsider what’s important

How we have worked to design EMPIRE for this platforms
- Specific instances

Results

Summary

Walk Down Memory Lane

First, I'll say it, I've been in HPC for a long while – yeah, I'm

Pre-1990 fastest computers were vector supercomputer
- Cray T90

Back in the 90's we were in a pretty famous parallel computing war
- Early in that decade Cray and vector machines ruled the days
- Around the middle of the decade lots of parallel computing languages
  - MPI won, PVM stuck around for a surprisingly long time
  - OpenMP starting in the 90s and is also around today, more on it later

We all know what happened next
- MPI ruled for the next 20ish years, and still does today, but it is not sufficient (MPI+X)

The shift then was computing power through more, less powerful, computers
- Every rank had their own memory and their own compute resources
- Clear memory ownership models existed

Top 500

The top supercomputing systems are tracked at http://top500.org
- Starting producing lists in 1993
- Judged by a synthetic benchmark – LinPack
- Top 500 today often becomes workhorse in 3 years

Notable favorites or disasters
- First one the CM5 1993
- ASCI Red – Cray XT3 1997-2000
  - Pretty much what we have today
- Roadrunner – 2008-2009
- Titan – 2012
  - First GPU system
- Summit – 2019

I'd argue
- Since 2000 almost all of the commodity systems look like the CM5

Currently 6 of the top 10 machines are GPU based
- Greater than 90% of Summit comes from the accelerators

Announced future machines slated for the top will have an accelerator

What is a Next Generation Platform

Next Generation Platforms (NGP) is a buzz word which has changed throughout the computing era

- It is a platform which looks different than current technology

Today NGP means something like

- An accelerator like a GPGPU
- A ton of concurrency
- Limited instruction sets
- Single program counter

Equally important

- Different programming models
  - OpenMP – Cuda – HIP – ROCm

What comes along with this

- New exciting ways to shoot yourself in your foot

Why Would I (you) Care?

(Q) All the standard machines run my legacy code, why would I care?

(A) If today's machines give you the turn around that you need and you'll never want more you shouldn't care

(Q) Look at road runner, it was a waste of time to port codes to that as it was here and gone?

(A) Seven years ago when machines like Titan were bleeding edge, GPU's were not a sure thing. However, future systems will have similar or different offload models than Summit.

(Q) Won't compilers save us?

(A) You mean like they did for auto-parallelism???

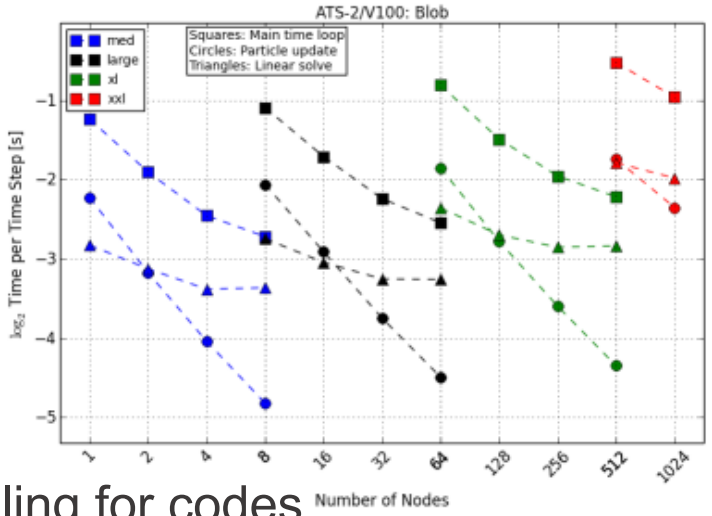
(Q) Won't libraries save us

(A) They'll help

- COPA Co-design center for Particle Applications
  - https://www.exascaleproject.org/ecp-co-design-center-looks-particle-based-applications-exascale/
- Trilinos for solvers

What Makes Next Gen Hard?

Characteristics of PIC



Plasma simulations, like everything, have a wide range of time scales
- Explicit and implicit methods allow you to only step over so much
- Unlike many disciplines, PIC requires order of a million steps

There are only 86400 seconds in a day
- If you have to complete millions of steps, you will need many steps/second
- Any small, constant slowdown will kill your scaling

Weak. Strong, and algorithmic scaling is typically how we measure scaling for codes
- Strong scaling - Double the number of “cores” achieve X% speed up (Matt’s def 33% faster you still “scaling”)
  - 46 more compute core for a 10x speedup
- Weak scaling holds the work constant per core and increases c
- Algorithmic is theoretical and what weak should be measured a

Almost all weak scaling plots for time accurate codes are lie
- Like the one to the upper right that I make
- Most make the problem bigger in number of mesh elements
  - Hold number of steps the same

True weak scaling
- Halve the mesh length, halve the timestep 16x the number of compute units
- As you weak scale, you are also strong scaling at the same time – looks bad typically
- Requires explanation, will be taken out of context, and you’ll be the laughing stock somewhere…

Scaling and Performance

Analysists don't care about scaling, they want a given quality answer in a given time
- Algorithmic performance most important – skipping that topic
- Weak scaling is an anti-performance measure. Want a perfectly weak scaling code on the full machine?
  - Insert sleep(100000); in your time loop, done!
  - If your multigrid solver weak scales **better** than NlogN, it has performance **problems**!
- Obviously that is a bad idea, but single node performance is much more important than weak scaling
  - Removing the line that says sleep(100000); in your time loop – that's single node performance in a nutshell
- Strong scaling is somewhere in the middle
  - Bad single node performance simplifies strong scaling
  - Perfect strong scaling codes are easy, switch to double double double quad math….

Matt's interpretation of Amdahl's law
- As you improve your code's single node performance, it will scale worse –or--
- If you find a way to do the math much faster, all you'll be left with is overhead

NGPs FLOPs are free, on device memory transfer is nearly free
- Off device transfer is expensive
- MPI is more expensive
- Allocations are expensive

In the NGP world we will focus on an MPI+X solution
- How do we design performance given X

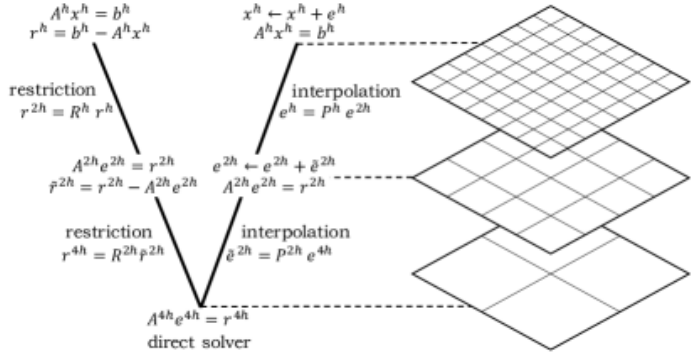
Differences in Next Generation – Scalar Performance

Performant codes for today's architectures are optimized for cache usage

- Memory latency is a critical bottleneck

NGP has all the current latency issues, but deeper memory hierarchie

- Take a GPGPU
  - More hierarchies on the card
  - Transfers off the cards – Like the old days with using disk space as RAM
  - Launch overhead – 10 microseconds to do 1 or 1million x=x+y;



Multigrid is the fastest ways to precondition Poisson's equation in terms of complexity - NlogN

- And on a yesterdays hardware, it most likely is

Smoothers are non-trivial – Gauss Seidel and ILU for example

- High concurrency mean coloring – launch overhead will kill you and you might not saturate the GPU

Coarse meshes are small, that's the logN

- Coarse meshes don't saturate the card
- Well, let's do a direct solve earlier – LU decomp is hard on GPGPUs

Jacobi is the stupidest preconditioner – however, it often is the best on GPGPUs

What Has EMPIRE Done to be Performant?

What is EMPIRE?

EMPIRE is a “new” code started in 2015 as part of the DoE’s ATDM program element

- It was initially created to determine can we build a next generation plasma code, using components in Trilinos to achieve a performant portable code
- Has the lofty goal of solving plasma problems across a large range of density ranges on unstructured meshes

EMPIRE-PIC is used for low density plasma simulations

- Largest scale calculation to date – 21B elements 1T particles

Built to be scalable from the start

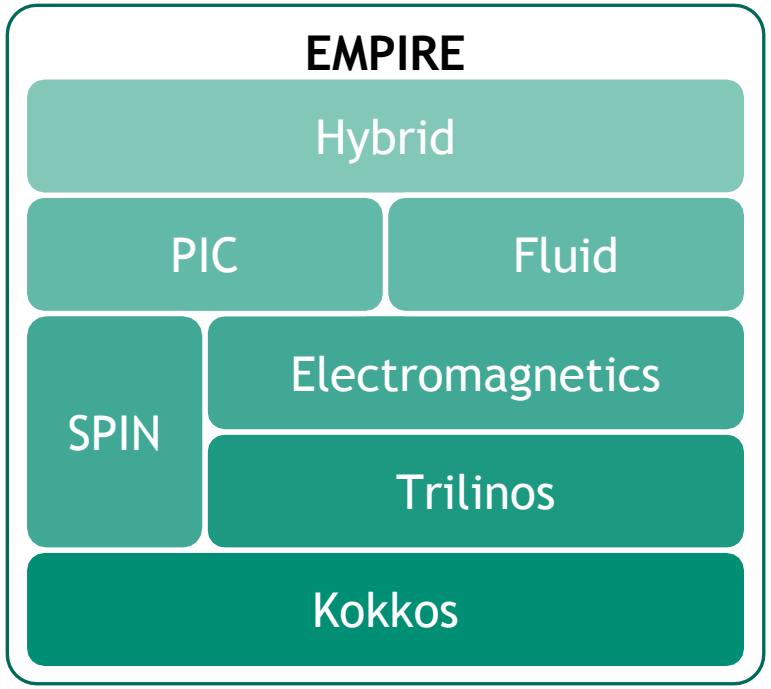
Electrostatic and Electromagnetic Maxwell solvers

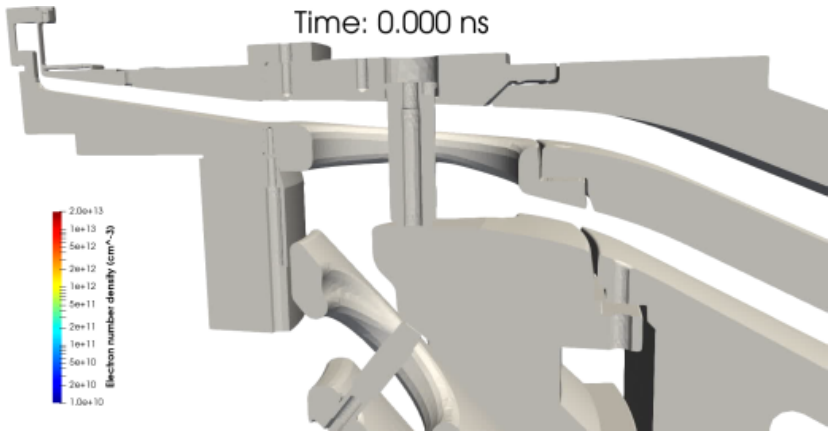
DSMC and MCC collisional models

Has an over-decomposition based load-balancing

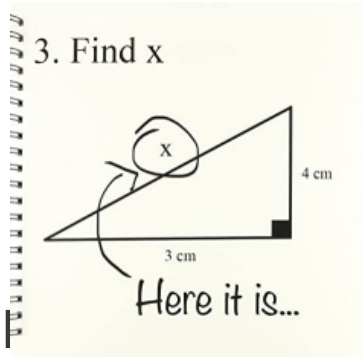
Simulation in the lower right of the Z-machine power flow

- Made by David Sirajuddin





What is X in MPI+X for EMPIRE

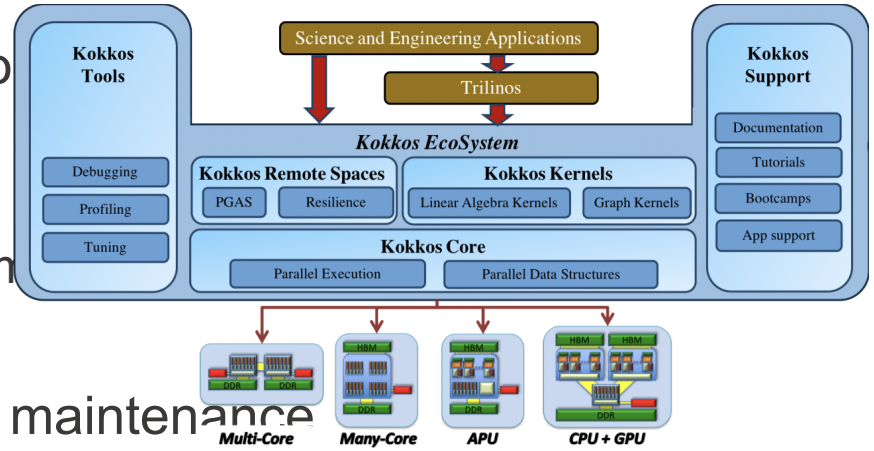


Stated previously we are designing for platforms which we know lots to little al

- Going to a MPI+X model. MPI for all the cross node parallelism and X for all the on node parallelism

Current platforms X is Cuda or OpenMP, but many more are co

- We are using Kokkos to abstract X to a single interface
- Kokkos inserts optimizations for the specific backends
- Kokkos is working with the C++ standards committee to get progran models into the C++ standard



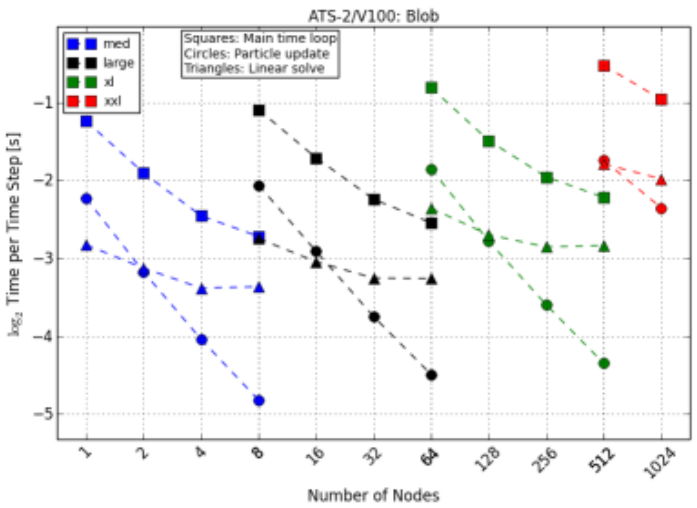
One programming model to program to code to simplifies code maintenance

- EMPIRE has **no** backend specific code in it

Lot's of good Kokkos info here

- https://github.com/kokkos/kokkos-tutorials/wiki/Kokkos-Lecture-Series
- https://github.com/kokkos/kokkos/wiki

Where Does the Time Go?



We have synthetic and real benchmarks which we have tested

- Time split between linear solve, particle update and diagnostics
- Particle update roughly 150M particles/second/GPU and strong scales
- Linear solve roughly 2M unknows/second/GPU and doesn't strong scale
  - Insufficient unknowns to scale well
- Diagnostics a small cost for all the problem currently

EMPIRE was tested and compared against our legacy code (EMPHASIS-64) on a real problem

- Medium size problem on Astra – SNL's Arm cluster with 640 nodes
- EMPIRE ran with **more** particles

	EMPHASIS-64bit	EMPIRE	Improvement
Total time stepping	20928	7286	2.9x
Linear Solve	6434	5018	1.3x
Particle Update	7308	1472	5.0x
Diagnostics	4195	451	9.3x

How Did We Get Here?

EMPIRE is drastically faster than our legacy code EMPHASIS – why, they have the same algorithms?

- Focusing on what matters – Single node performance
- All of this is specifically for particles but true for diagnostics and solvers

Data models are critical

- While NGP have faster memory access, one still has to be careful
- AoS and SoA used to be the question people would argue over – EMPIRE uses SoSoAoS

SoSoAoS

- Core structure is filled with DEQue like structures, one each for velocity, position, types, ….
  - Allows an routine to only access what it needs
- The DEQue like structure is a stack of arrays where fixed sizes arrays can be pushed/popped
  - Allows for constant time memory growth/reduction
  - Access uses shift operators – cheap contiguous access
- The arrays hold a single component (position) for a chunk of particles

Particles are marked for deletion and then removed later

- Contiguous memory access for better performance

Atomic operations are available for parallel lock free addition and deletion of particles

Memory pools are used for all temporaries and recycled – allocations are slow

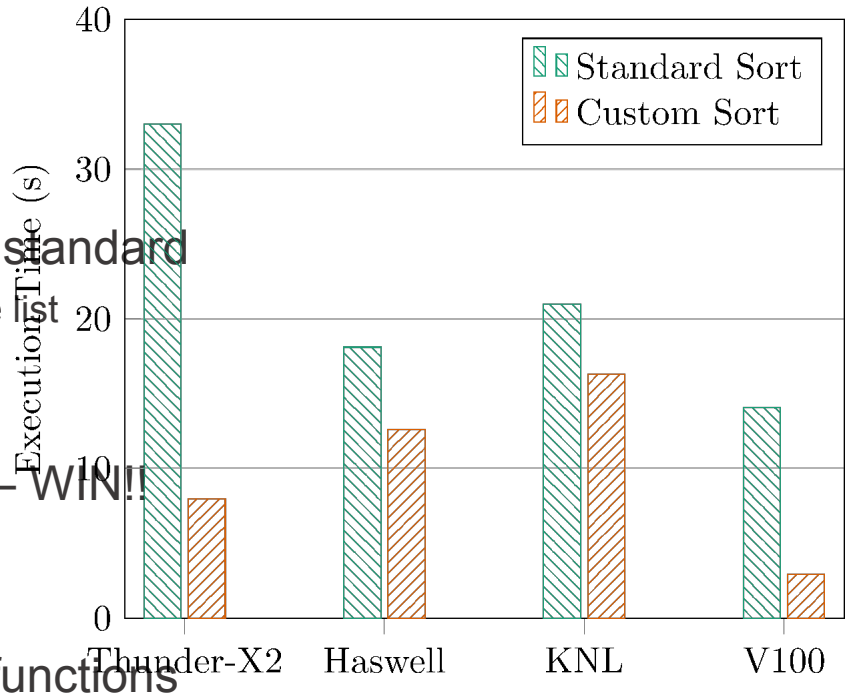
Good Data Structures Need Good Algorithms

Let's explore one part of the particle update – Boris acceleration

- Loop through particles gather E/B from the mesh to a temporary on the particle and then apply F=mA
- Good data structures mean you only need to bring in solver data, position, velocity and type info
- Sorting the particles will yield good memory reuse
  - Only load the element data once for all the particles within the element

Sorting is expensive

- Thrust offers a good sort with order N cost, often viewed as the gold standard
  - Thrust would take element and type arrays and determine where a particle lives in the list
  - Then we would move the data in parallel into a temporary and then back
  - Only moves the data twice versus logN in something like quicksort
- The cost of the sort was more than offset by the particle push gains – WIN!!
  - The sort was still expensive
- Develop a custom sort which was much faster than Thrust
- This allows us to sort more often and improve performance in other functions



Wash, Rinse, Repeat

- Explore every kernel - Develop kernel performance test, optimize, insert

Summary

Ranking importance of performance
- Algorithmic – Single node – Strong scaling – Weak scaling

We need to rethink scaling and performance
- Scaling without single node performance has been the mantra for a long time
- Single node performance is key
- Strong scaling is hard for codes that have good single node performance
- Weak scaling is only useful for estimating runtime for big runs, but is mostly pointless
  - More of a hardware metric or an idiot check if your code is reasonable

Single node performance is key (I know I said that above)

Understanding data structures is critical for single node performance
- Have I mentioned that it is key?

Libraries can be good for performance and portability
- Libraries, even the best, can really slow down your code or prevent other optimizations