On the Origin of the Programming-Models

Tim Mattson

Intel, Parallel Computing Lab

This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

Legal Disclaimer

Human Readable Disclaimer

- The views expressed in this talk are those of the speaker and not his employer.
- These slides are borrowed from other talks and other people. Sorry if you've heard much of this before. But remember, I'm talking as an HPC historian in this talk, and we know that …
  - While history repeats itself, historians often repeat each other.

I work in Intel's research labs. I don't build products. Instead, I get to poke into dark corners and think silly thoughts… just to make sure we don't miss any great ideas.

Hence, my views are by design <u>far</u> "off the roadmap".

The quest for the "right" Programming model

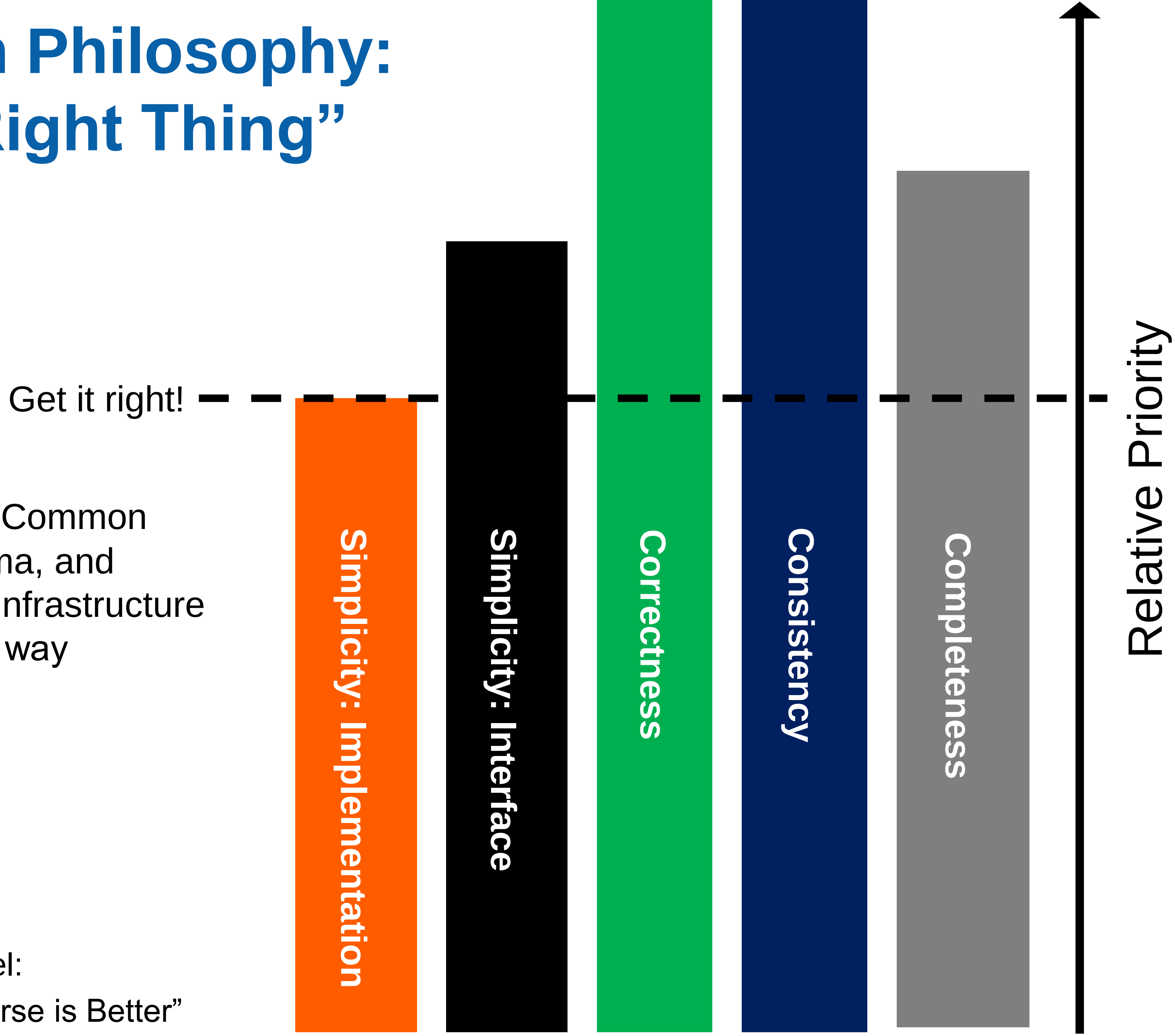
- I am a molecular physicist …. I don't even like computers ... So its ironic that I am best known for my work on programming models:

Ada	mid-80's	DOD mandated object based language	Complex, big, and nobody likes to be forced to use a new language
Strand	Late 1980's	Concurrent logic programming	Failed miserably … nobody wanted to lear a new language
Linda	Late 1980's to mid-90's	Coordination language built around a tuplespace	Failed miserably … why pay money when you can get something (PVM) that works for free?
MPI	mid-90's to current	Roll the best message passing ideas into one system	MPI is the most successful parallel programming language ever.
OpenMP	Late-90's to present	Shared memory programming made simple	The second most successful parallel programming language.
OpenCL	2007 to present	Portable platform for heterogeneous systems	Application programmers screwed up … long live CUDA
OCR	2010 to present	Assync. Multi-tasking.	To early to say ….

Making sense of the programming models

- To understand which programming models succeed an which fail, we need to start with the famous essay by Richard Gabriel … “The rise of worse is better”
  - An essay that tried to explain the failure of common LISP to become a dominant programming model.

Design Philosophy: "The Right Thing"



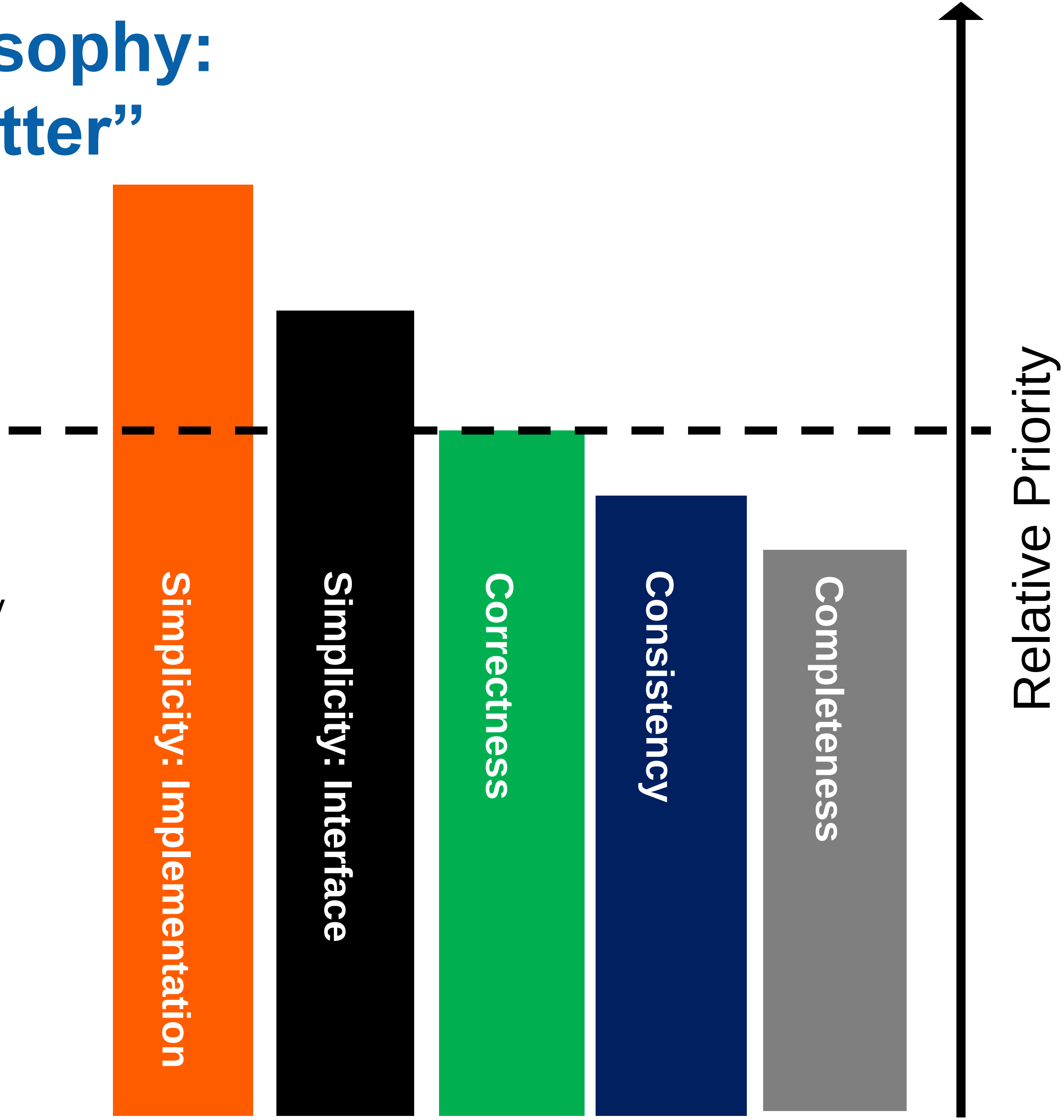
Example: Common Lisp, Schema, and supporting infrastructure … The MIT way

Richard Gabriel:

The rise of Worse is Better"

"https://www.jwz.org/doc/worse-is-better.html

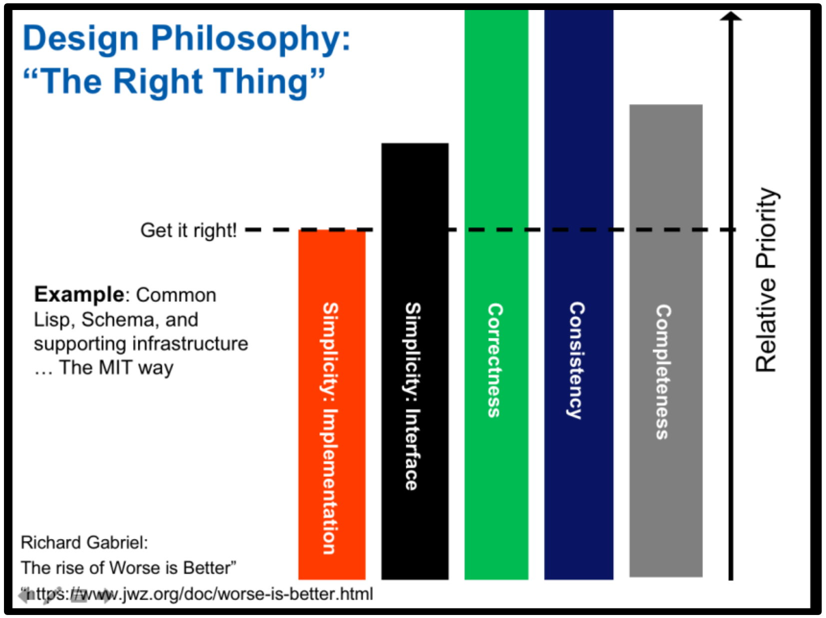
Design Philosophy: "Worse is Better"

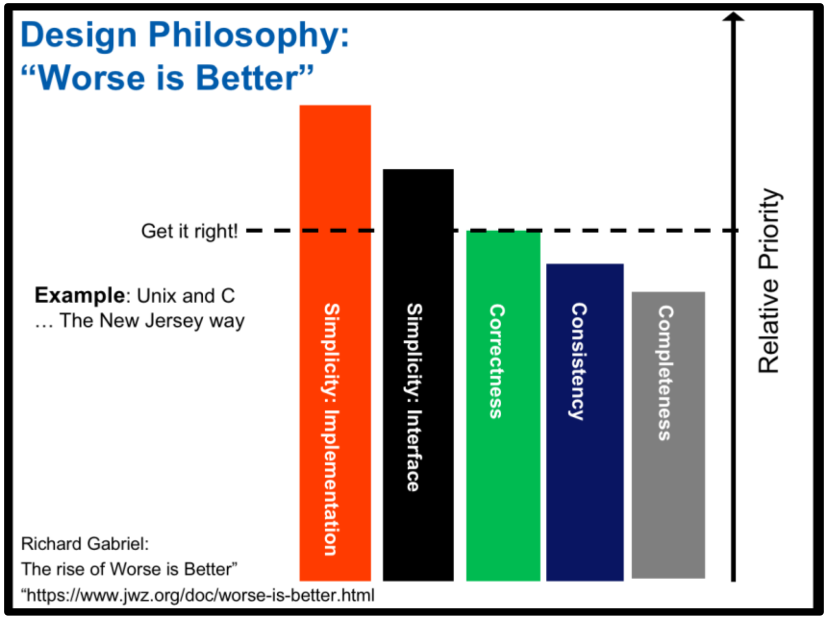


Example: Unix and C
… The New Jersey way

Richard Gabriel:
The rise of Worse is Better"
"https://www.jwz.org/doc/worse-is-better.html

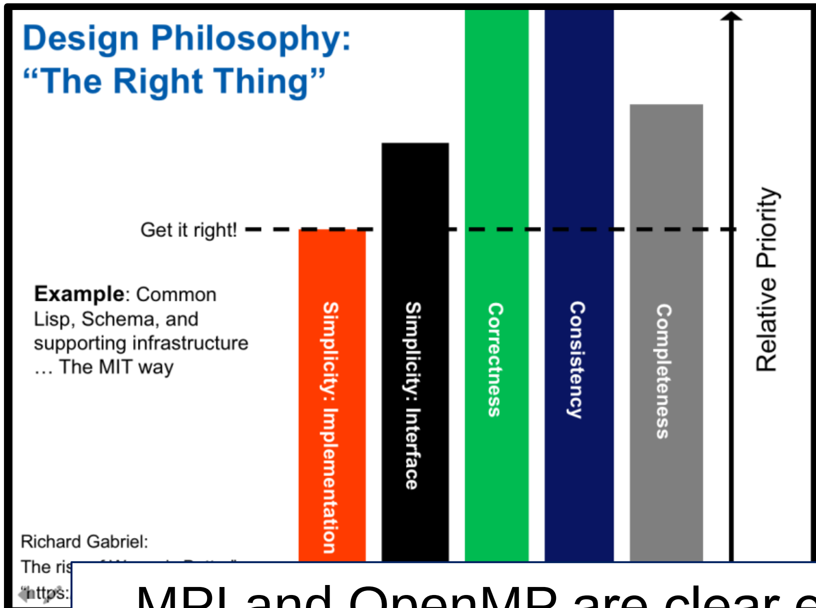
Which Design Philosophy wins?

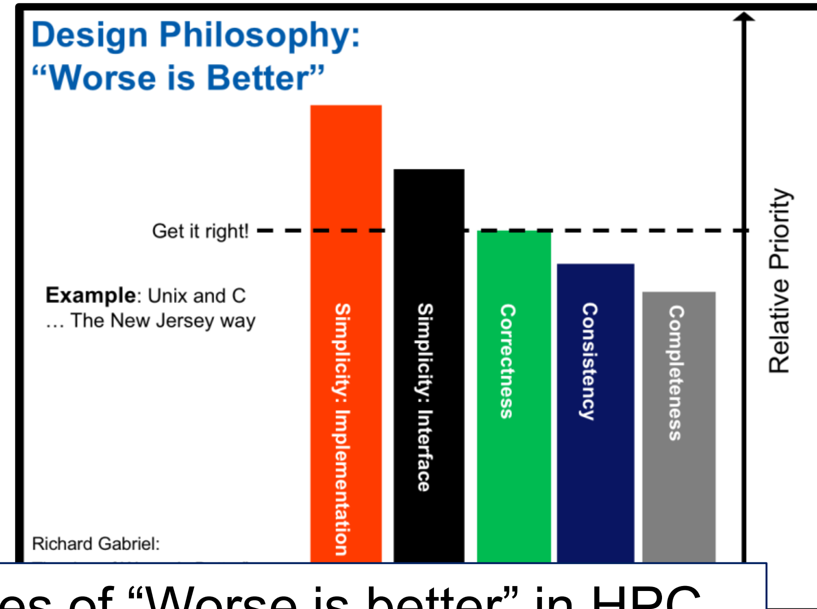




- History shows again and again … “Worse is better”.
  - While “the right thing” take the time to “get it right”, the “worse is better” folks are busy establishing a user base.
  - “Worse is better” programmers are conditioned to sacrifice safety, convenience, and hassle to get good performance.
  - Since “worse is better” stresses implementation simplicity, its available everywhere.
  - With a large user base, once “worse is better” has spread, there is pressure to improve it … so over time it becomes good enough

Which Design Philosophy wins?





MPI and OpenMP are clear examples of “Worse is better” in HPC

- History shows again and again … “Worse is better”.
  - While “the right thing” take the time to “get it right”, the “worse is better” folks are busy establishing a user base.
  - “Worse is better” programmers are conditioned to sacrifice safety, convenience, and hassle to get good performance.
  - Since “worse is better” stresses implementation simplicity, its available everywhere.
  - With a large user base, once “worse is better” has spread, there is pressure to improve it … so over time it becomes good enough

But there must be better ways to program parallel computers!

Parallel programming environments in the 90's

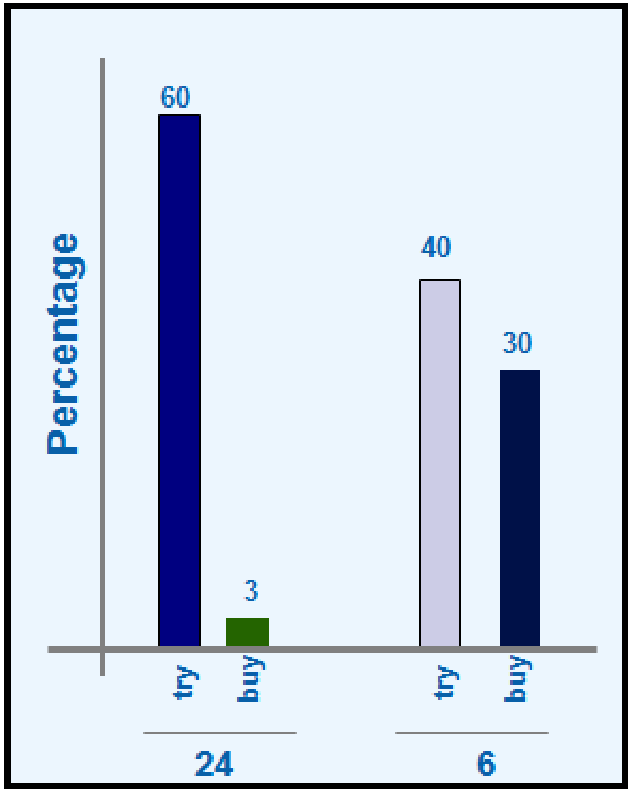
ABCPL
ACE
ACT++
Active messages
Adl
Adsmith
ADDAP
AFAPI
ALWAN
AM
AMDC
AppLeS
Amoeba
ARTS
Athapascan-0b
Aurora
Automap
bb_threads
Blaze
BSP
BlockComm
C*.
"C* in C
C**
CarlOS
Cashmere
C4
CC++
Chu
Charlotte
Charm
Charm++
Cid
Cilk
CM-Fortran
Converse
Code
COOL
CORRELATE
CPS
CRL
CSP
Cthreads
CUMULVS
DAGGER
DAPPLE
Data Parallel C
DC++
DCE++
DDD
DICE.
DIPC
DOLIB
DOME
DOSMOS.
DRL
DSM-Threads
Ease .
ECO
Eiffel
Eilean
Emerald
EPL
Excalibur
Express
Falcon
Filaments
FM
FLASH
The FORCE
Fork
Fortran-M
FX
GA
GAMMA
Glenda
GLU
GUARD
HAsL.
Haskell
HPC++
JAVAR.
HORUS
HPC
IMPACT
ISIS.
JAVAR
JADE
Java RMI
javaPG
JavaSpace
JIDL
Joyce
Khoros
Karma
KOAN/Fortran-S
LAM
Lilac
Linda
JADA
WWWinda
ISETL-Linda
ParLin
Eilean
P4-Linda
Glenda
POSYBL
Objective-Linda
LiPS
Locust
Lparx
Lucid
Maisie
Manifold
Mentat
Legion
Meta Chaos
Midway
Millipede
CparPar
Mirage
MpC
MOSIX
Modula-P
Modula-2*
Multipol
MPI
MPC++
Munin
Nano-Threads
NESL
NetClasses++
Nexus
Nimrod
NOW
Objective Linda
Occam
Omega
OpenMP
Orca
OOF90
P++
P3L
p4-Linda
Pablo
PADE
PADRE
Panda
Papers
AFAPI.
Para++
Paradigm
Parafrase2
Paralation
Parallel-C++
Parallaxis
ParC
ParLib++
ParLin
Parmacs
Parti
pC
pC++
PCN
PCP:
PH
PEACE
PCU
PET
PETSc
PENNY
Phosphorus
POET.
Polaris
POOMA
POOL-T
PRESTO
P-RIO
Prospero
Proteus
QPC++
PVM
PSI
PSDM
Quake
Quark
Quick Threads
Sage++
SCANDAL
SAM
pC++
SCHEDULE
SciTL
POET
SDDA.
SHMEM
SIMPLE
Sina
SISAL.
distributed smalltalk
SMI.
SONiC
Split-C.
SR
Sthreads
Strand.
SUIF.
Synergy
Telegrphos
SuperPascal
TCGMSG.
Threads.h++.
TreadMarks
TRAPPER
uC++
UNITY
UC
V
ViC*
Visifold V-NUS
VPE
Win32 threads
WinPar
WWWinda
XENOOPS
XPC
Zounds
ZPL

A warning I've been making for the last 10 years

Is it bad to have so many languages?

Too many options can hurt you

- The Draeger Grocery Store experiment consumer choice:
  - Two Jam-displays with coupon's for purchase discount.
    - 24 different Jam's
    - 6 different Jam's
  - How many stopped by to try samples at the display?
  - Of those who "tried", how many bought jam?



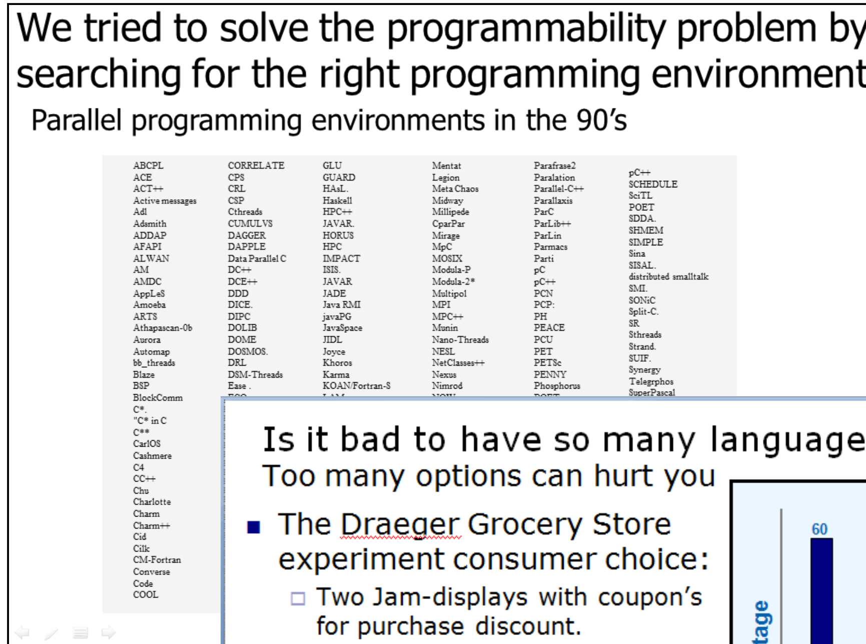
Programmers don't need a glut of options … just give us something that works OK on every platform we care about. Give us a decent standard and we'll do the rest

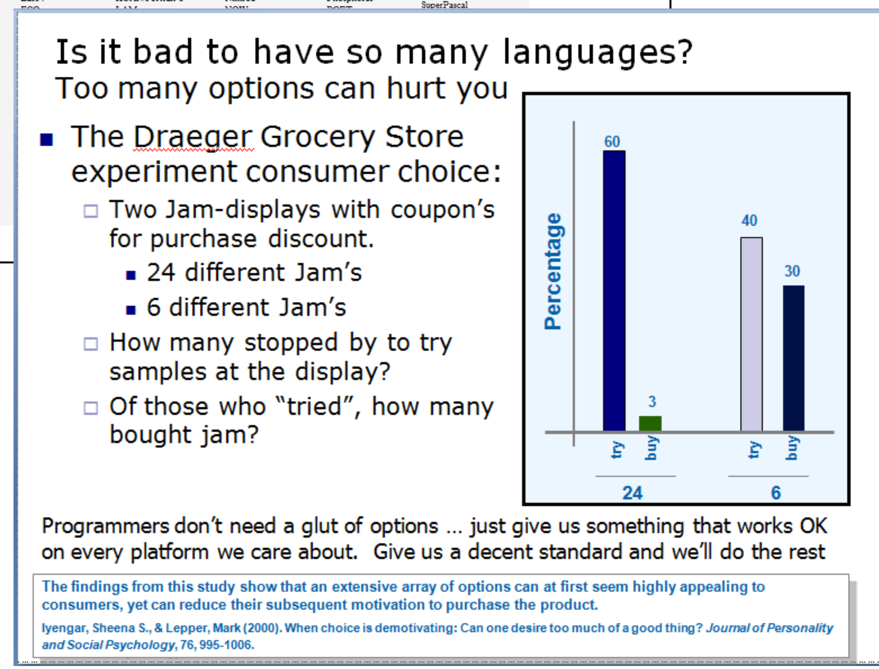
The findings from this study show that an extensive array of options can at first seem highly appealing to consumers, yet can reduce their subsequent motivation to purchase the product.

Iyengar, Sheena S., & Lepper, Mark (2000). When choice is demotivating: Can one desire too much of a good thing? *Journal of Personality and Social Psychology*, 76, 995-1006.

But this isn’t quite fair

- The glut of parallel languages and choice overload matter a great deal when your job is to reach out to application programmers and software vendors.
- But is it really something Computer Science researchers should be overly concerned about?



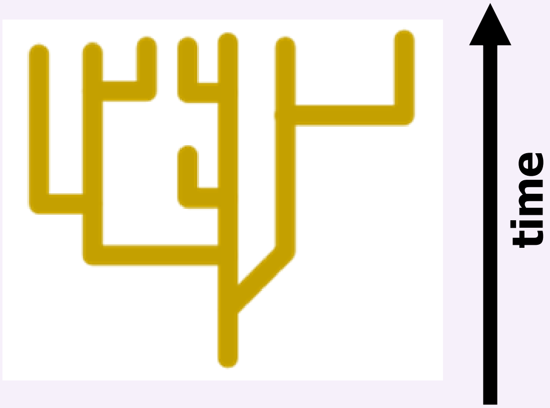


We need a more nuanced way to think about the role of academic computer science research

Evolution

- Two Major Models of evolution:
  - **Phyletic Gradualism**:
    A slow, continuous and gradual process of change.
  - **Punctuated Equilibrium**:
    Long periods where little change is observed interspersed with periods of abrupt change.

How these trends would appear in the fossil record



Source: http://en.wikipedia.org/wiki/File:Punctuated-equilibrium.svg

What actually happens in biological Evolution?

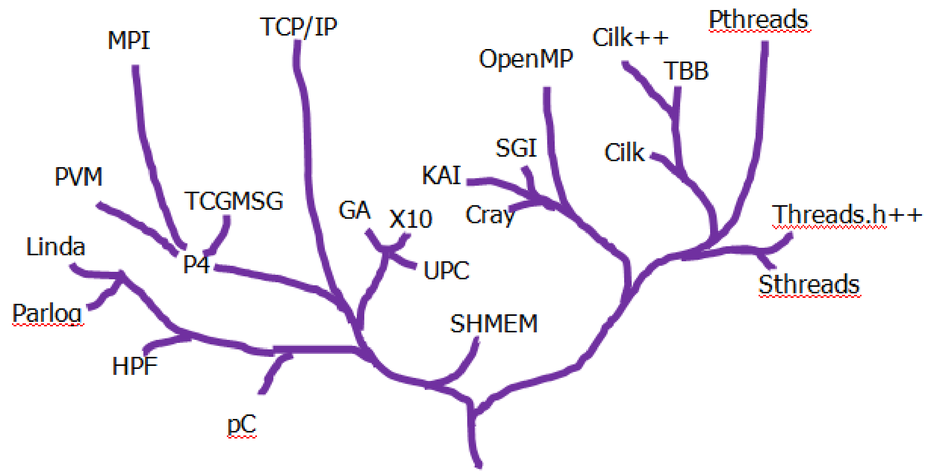
What actually happens in Biology?

- Both Phyletic Gradualism and Punctuated Equilibrium occur:
  - Decompose the ecosystem into relatively isolated niches.
  - Inside an isolated niche, phyletic Gradualism occurs
  - An event changes the ecosystem (e.g. climate change) so what was once an obscure Niche becomes the new mainstream.
  - Since an adaptive species evolved to match the “new mainstream” in a protected niche, it is now well positioned to dominate the new normal.
- So the answer of phyletic gradualism vs. punctuated equilibrium is “both”.

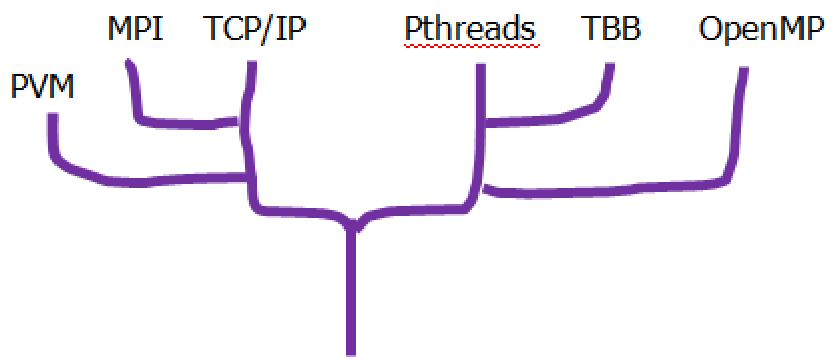
How does this apply to programming models?

Evolution of parallel programming environments

- Both Models are observed depending on the ecosystem:
  - **Research community**
    - Phyletic Gradualism with lateral gene transfer:
      (1) large, growing population of parallel programming systems,
      (2) occupy isolated environmental niches, and (3) few real users.



  - **Professional Application developers**
    - Punctuated equilibrium …
      (1) a small number of stable programming systems that
      (2) change rarely in response to abrupt external forces.

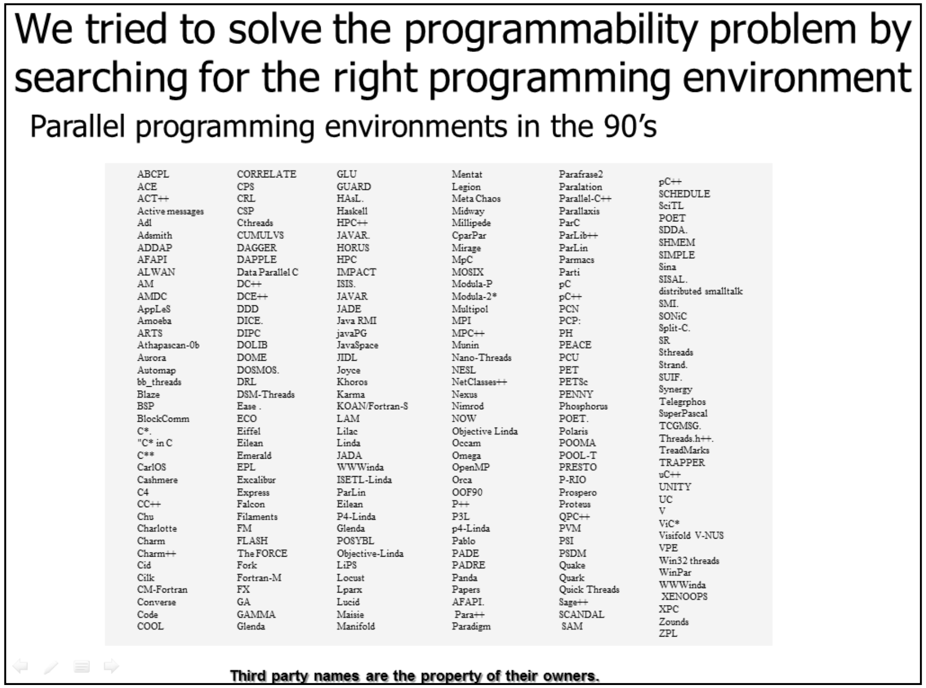


Computer Science research creates the innovation so new "species" are ready when environmental conditions change and new "mainstream" systems emerge.

Third party names are the property of their owners

External change and the evolution

- External changes establish a "new normal" and fringe academic projects walk in to dominate.
- So It's a "good thing" that researchers are creating so many programming languages … as long as they don't prematurely push them into the mainstream (choice overload).



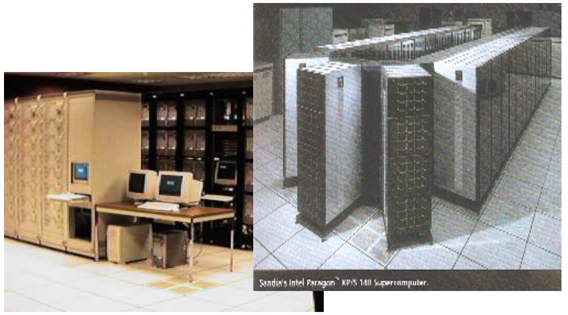
- To understand the future of computing we need to look at the key changes (or inflection points) just around the corner
- And then ask … which academic/research projects are "in the wings" waiting to emerge and dominate the new normal

Key HPC hardware inflection points

- The first multiprocessor: Burroughs B5000, 1961
- SMP goes mainstream: the Intel® Pentium™ technology in 1995 (up to two processors) and the Intel® Pentium Pro® processors (up to four).

Dual socket Pentium pro board (~1997)

- MPPs (e.g. Intel ® Paragon, TMC CM5, Cray T3D) in early 90’s,
- Clusters (Stacked Sparc pizza boxes late 80’s) and Linux clusters starting with Beowulf in 1994.



NCSA super-cluster (1998) and Paragon XPS 140 (1994)

- GPGPU programming starts in early 2000’s but using primitive shader language
- NVIDIA innovations lead to fully programmable GPUs

NVIDIA GeForce 8800/HD2900 (~2006)

Key HPC hardware inflection points

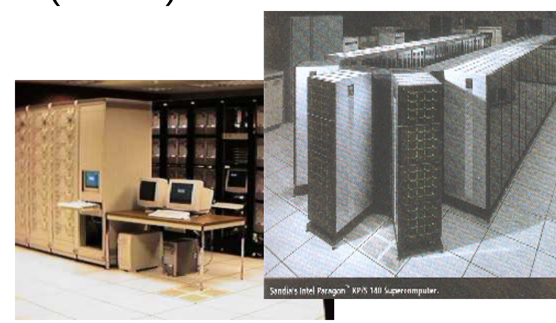
- The first multiproces ... 961
- SMP goes mainstre ... nnology in 1995 (up to two proc ... Pro (up to four processors).

OpenMP

Dual socket Pentium pro board (~1997)

- MPPs (e.g. Paragon, ... CM5, Cray T3D) in early 90's,
- Clusters (Stacked Sparc pizza boxes late 80's) and Linux clusters starting with Beowulf in 1994.

MPI



NCSA super-cluster (1998) and Paragon XPS 140 (1994)

- GPGPU programm ... ly 2000's but using pri ... er lan
- NVIDIA innovations ... ogrammable GPUs

CUDA

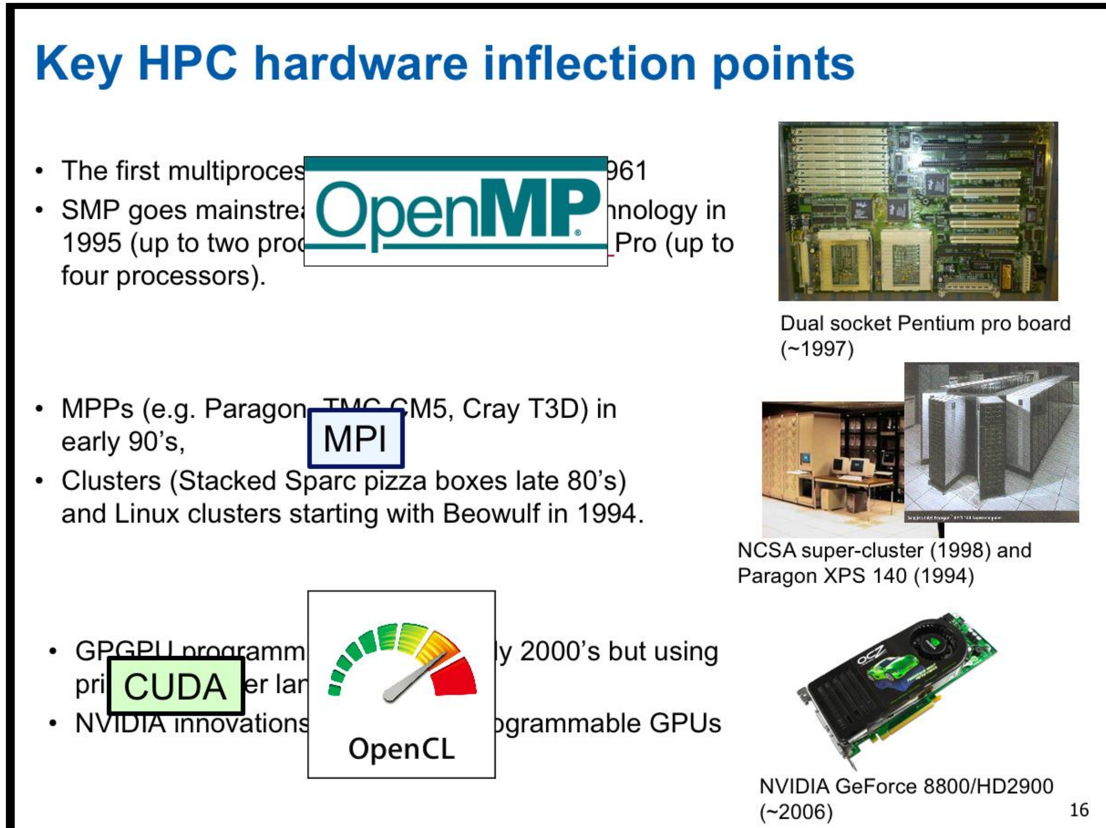
OpenCL

NVIDIA GeForce 8800/HD2900 (~2006)

Third party names are the property of their owners

Adoption of HPC programming models

- New HPC programming models only occur at hardware inflection points
- In between inflection points, it is more productive to improve existing models than to push new ones.

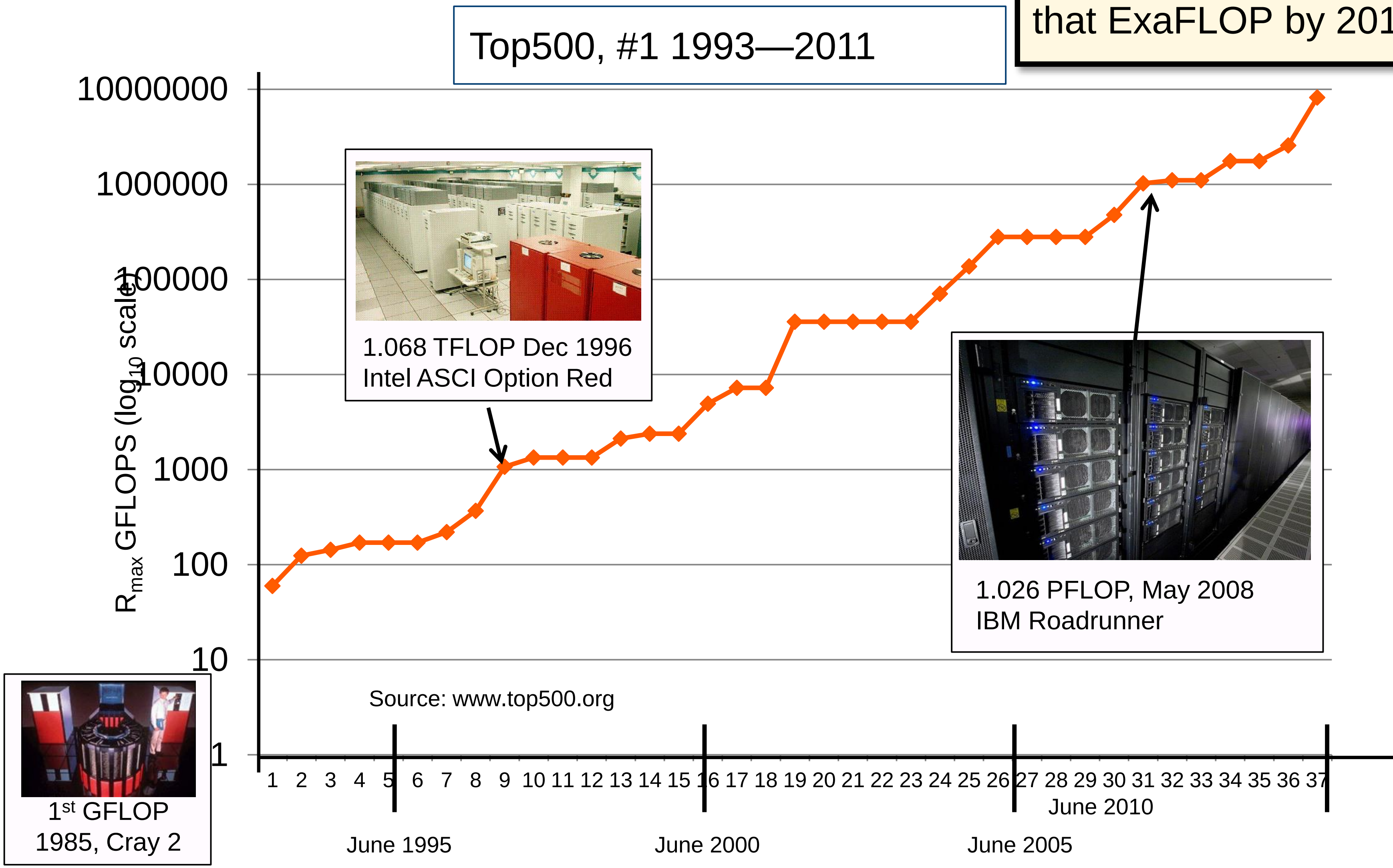


Are there any HPC inflection points looming on the horizon?

Maybe … consider Exascale computing

A Linpack ExaFLOP

If we stick to historical trends, we should have that ExaFLOP by 2018

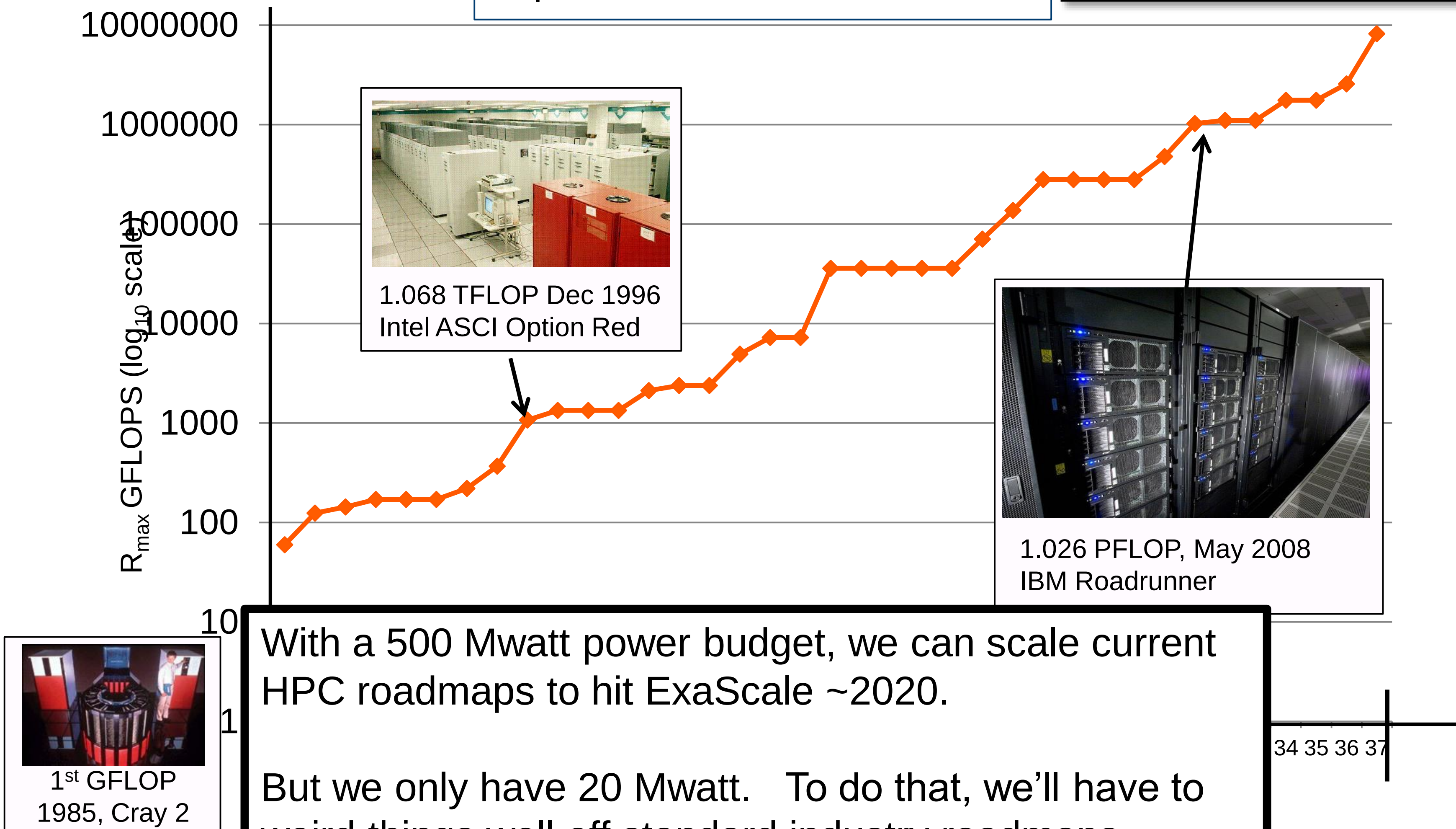


So how are we doing? Are we on track to hit our goal?

A Linpack ExaFLOP

If we stick to historical trends, we should have that ExaFLOP by 2018

Top500, #1 1993—2011



1st GFLOP
1985, Cray 2

With a 500 Mwatt power budget, we can scale current HPC roadmaps to hit ExaScale ~2020.

But we only have 20 Mwatt. To do that, we'll have to weird things well off standard industry roadmaps.

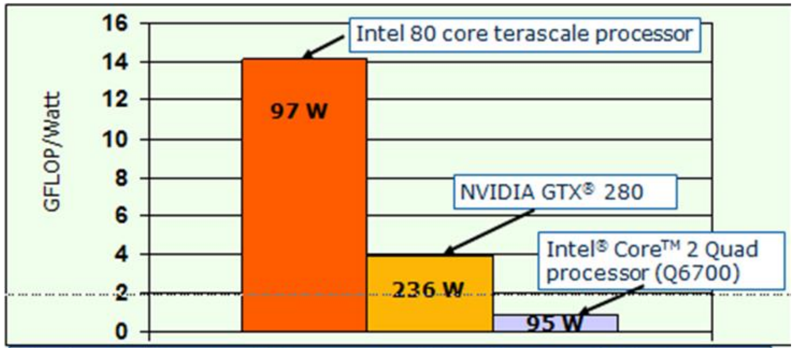
The ExaScale Challenge: Pity the Poor Programmer

- Vast numbers of low power cores

Capacitance = C
Voltage = V
Frequency = f
Power = CV^2f

- O(billion) parallelism. Remember Amdahl's law?

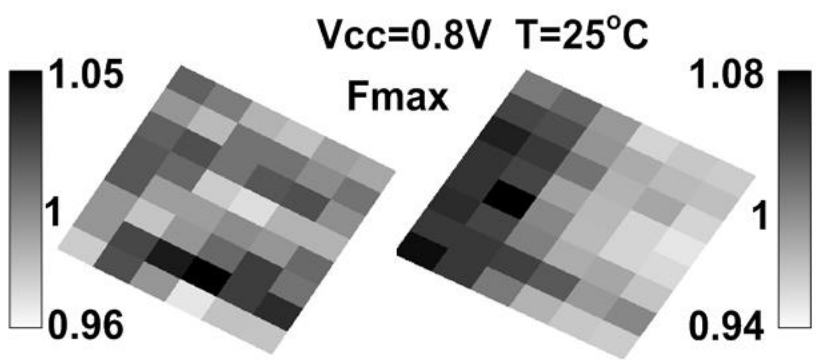
- A mix of general purpose, vector, and special function cores.



This is a common trend: the more specialized the core, the better the power/watt. This is why heterogeneous many core chips are inevitable.

- Parallel composition of SW specialized to different cores

- Near threshold logic at 7 nm (or lower) process technology



Freq. variation across a single many-core chip

- Variability, reliability, and silent errors

Probability of Faults

Fault	Probability	Impact	Action
Fans	High	Low	Node down
Power Supply	High	Low	Node down
CPU / SRAM	Very Low	Low	Node down
DRAM	Medium	Low	Reconfiguration
Solder Joints	High	Low	Node down
Sockets	High	Low	Node down
Disks	Mid to High	Low	Reconfiguration
NAND/PCM	Low	Low	Reconfiguration
Soft Errors	Low	High	Clever accounting

Source: John Daly, David Mountain's NSA Resiliency WS 02/2012

Source: Shekhar Borkar, Intel, Sept. 2014.

Probability of Faults

Fault	Probability	Impact	Action
Fans	High	Low	Node down
Power Supply	High	Low	Node down
CPU / SRAM	Very Low	Low	Node down
DRAM	Medium	Low	Reconfiguration
Solder Joints	High	Low	
Sockets	High	Low	
Disks	Mid to High	Low	
NAND/PCM	Low	Low	
Soft Errors	Low	High	accounting

Only a HW person would mark these low … to my app, these kill the program and therefore have a high impact

Source: John Daly, David Mountain's NSA Resiliency WS 02/2012

Source: Shekhar Borkar, Intel, Sept. 2014.

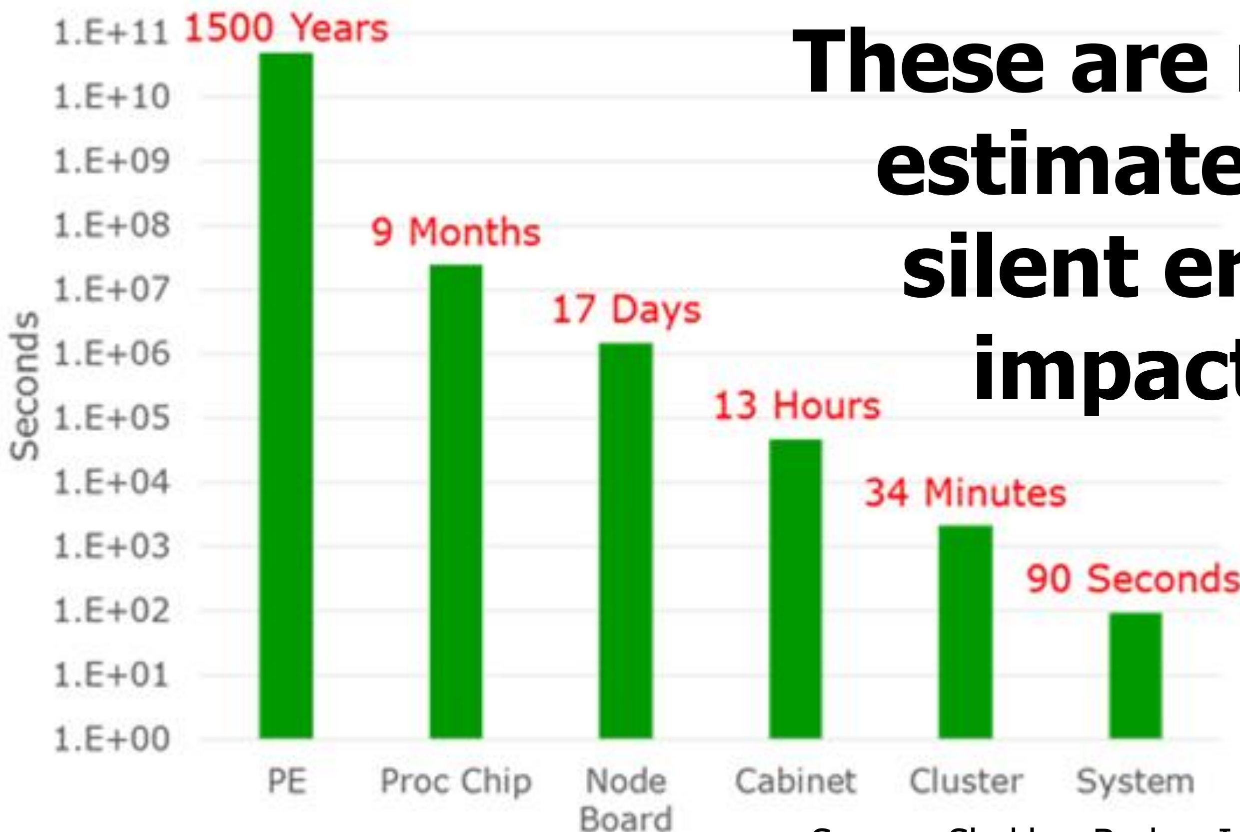
Probability of Faults

Fault	Probability	Application Software Impact	Action
Fans	High	**High**	Node down
Power Supply	High	**High**	Node down
CPU / SRAM	Very Low	**High**	Node down
DRAM	Medium	**High**	Reconfiguration
Solder Joints	High	**High**	Node down
Sockets	High	**High**	Node down
Disks	Mid to High	**High**	Reconfiguration
NAND/PCM	Low	**High**	Reconfiguration
Soft Errors	Low	**Terrifying**	Clever accounting

Source: John Daly, David Mountain's NSA Resiliency WS 02/2012

Source: Shekhar Borkar, Intel, Sept. 2014.

Mean Time to Error (Single Event)



These are rough estimates of silent error impacts

Source: Shekhar Borkar, Intel, Sept. 2014.

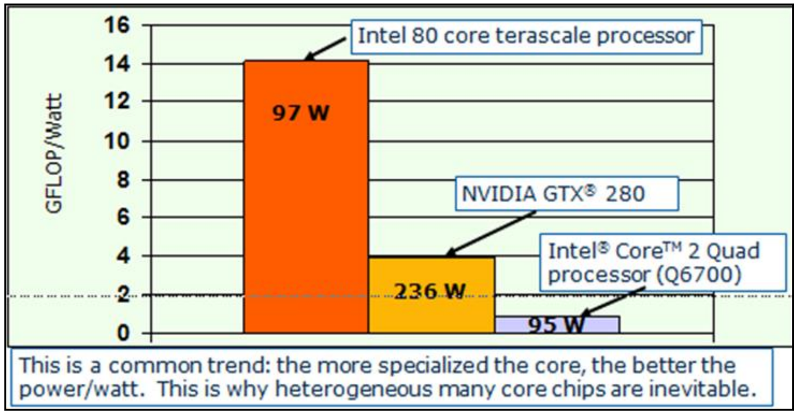
Select confinement and recovery (state restore) based on probability of error

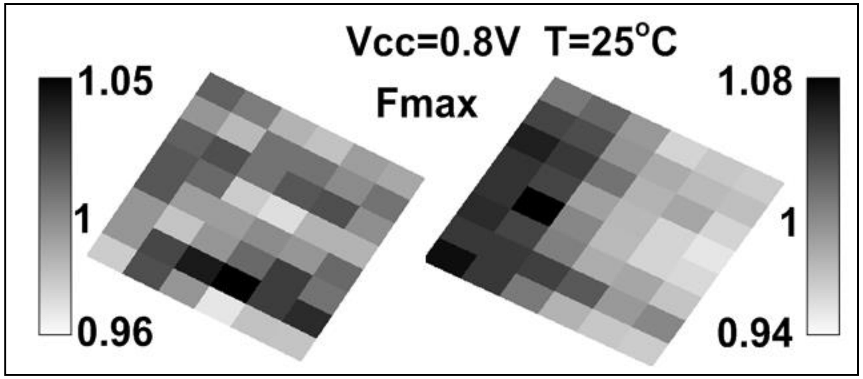
i.e. an exascale system is a neutron detector every 90 seconds

Assumptions, 10^{-4} soft error rate for an exaScale system with 2000 PE/cores per proc, 16 procs per node, 32 nodes per cabinet, 500 Cabinets per system

HW inflection points

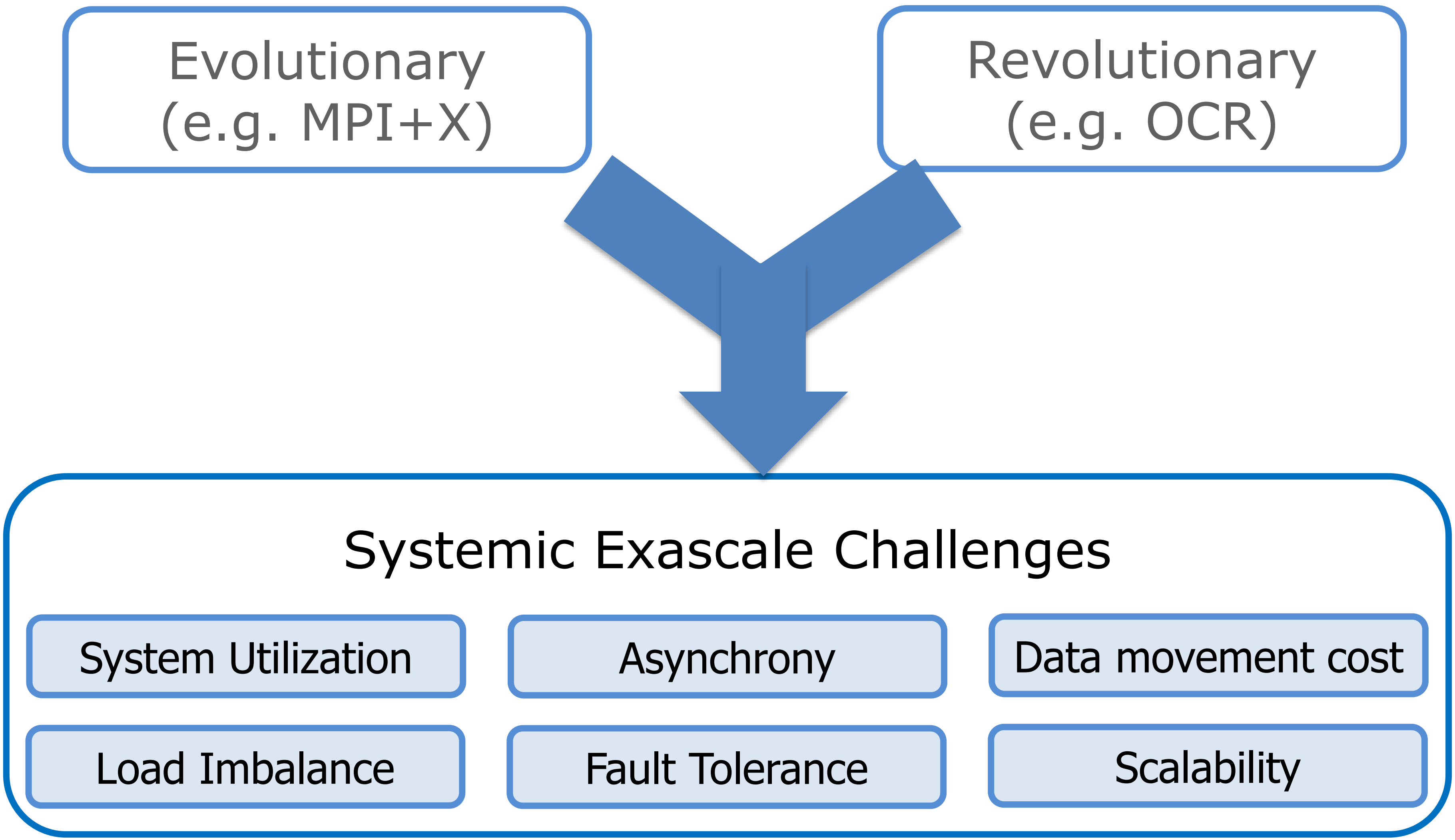
- Resilience + heterogeneity + extreme scale suggests exaScale will be a major hardware inflection point.





So maybe at last, we can realistically push something besides MPI+X into the HPC applications community?

2 pathways to Exascale Runtime Research



Third party names are the property of their owners

Exascale Runtime Candidates

Model	Memory	Load balancing: SMP	Load balancing: Between nodes	User interface	Execution model	Fault tolerance	Collectives
OCR	relocatable data blocks	work stealing	data block seeding	library	event driven tasks	----	barrier
Charm++	relocatable (arrays of) objects	migration	migration + object seeding	translator	asynch RMI	local checkpoint + message logging	barrier, bcast, reduce, chare array
Legion	relocatable hierarchical data blocks	work stealing	data block seeding + ?	translator or library	asynch RMI	----	barrier, reduce, task array
Grappa	PGAS	work stealing		library	data driven tasks	----	barrier, bcast, reduce, on-all-cores
HPX/ XPI	AGAS/PGAS	work stealing	---	library	message driven execution	----	barrier, reduce
MPI+X (X= MPI OpenMP)	Node-local + X	X	App-level	Library + X	CSP+X	UFLM	Full set

Exascale Runtime Candidates

Model	Memory	Load balancing		User interface	Execution model	Fault tolerance	Collectives
		SMP	Between nodes				
OCR	relocatable data blocks	work stealing	data block seeding	library	event driven tasks	----	barrier
Charm++	relocatable (arrays of) objects	migration	migration + object seeding	translator	asynch RMI	local checkpoint + message logging	barrier, bcast, reduce, chare array
Legion					asynch RMI	----	barrier, reduce, task array
Grappa	PGAS	work stealing		library	data driven tasks	----	barrier, bcast, reduce, on-all-cores
HPX/ XPI	AGAS/PGAS	work stealing	---	library	message driven execution	----	barrier, reduce
MPI+X (X= MPI OpenMP)	Node-local + X	X	App-level	Library + X	CSP+X	UFLM	Full set

Often referred to as Asynchronous Multi-tasking (AMT) systems

You can only understand a software system by using it.

- Ideal Case …
  - This is an important problem, so we must take as long as needed to get it right
  - You must work with full applications
  - Many problems only appear “at scale”

- The harsh byte of reality
  - Hardware is only a few years away and applications teams need to start soon to get ready. We need answers NOW!!!
  - Full applications take Person-YEARS to write/optimize for a new system.   With so many systems to study, we simply must find simpler test cases to use.
  - Full Scale systems are busy doing real work.

The Parallel Research Kernels (PRK)

PRK: Low level constructs that capture the essence of what parallel programmers require from parallel computers.

- **The PRK**
  - A set of low level “research kernels” to guide the design of future systems.
  - Output from full apps analysis overwhelming, hard to digest. The PRK can even be understood by HW engineers.
  - They are simple, small and easy to run … support simulator analysis or even “ paper and pencil” machines.

- **Selection Methodology**
  - Anecdotal analysis by a panel of experienced parallel application programmers at Intel.
  - Great Care was taken to make sure their experience covered the full range of typical HPC applications.

https://github.com/ParRes/Kernels

The Parallel Research Kernels version 1.0

- Dense matrix transpose
- Synchronization: global (collective) and point to point
- Scaled vector addition (Stream triad)*
- Atomic reference counting, both contended and uncontended (locks/TSX)
- Vector reduction
- Sparse matrix-vector multiplication
- Random access update
- Stencil computation
- Dense matrix-matrix multiplication (~DGEMM)
- Branch (inner-loop conditionals + PC panic)*

These are for "old school" HPC; can be statically load balanced. Summer'2016 we'll add a new case that demand dynamic load balancing (a Particle in Cell code)

**embarrassingly parallel*

The Parallel Research Kernels version 1.0

- **Dense matrix transpose**
- **Synchronization**: global (collective) and **point to point**
- Scaled vector addition (Stream triad)*
- Atomic reference counting, both contended and uncontended (locks/TSX)
- Vector reduction
- Sparse matrix-vector multiplication
- Random access update
- **Stencil computation**
- Dense matrix-matrix multiplication (~DGEMM)
- Branch (inner-loop conditionals + PC panic)*

Red Text indicates subset used in our exaScale studies

These are for "old school" HPC; can be statically load balanced. Summer'2016 we'll add a new case that demand dynamic load balancing (a Particle in Cell code)

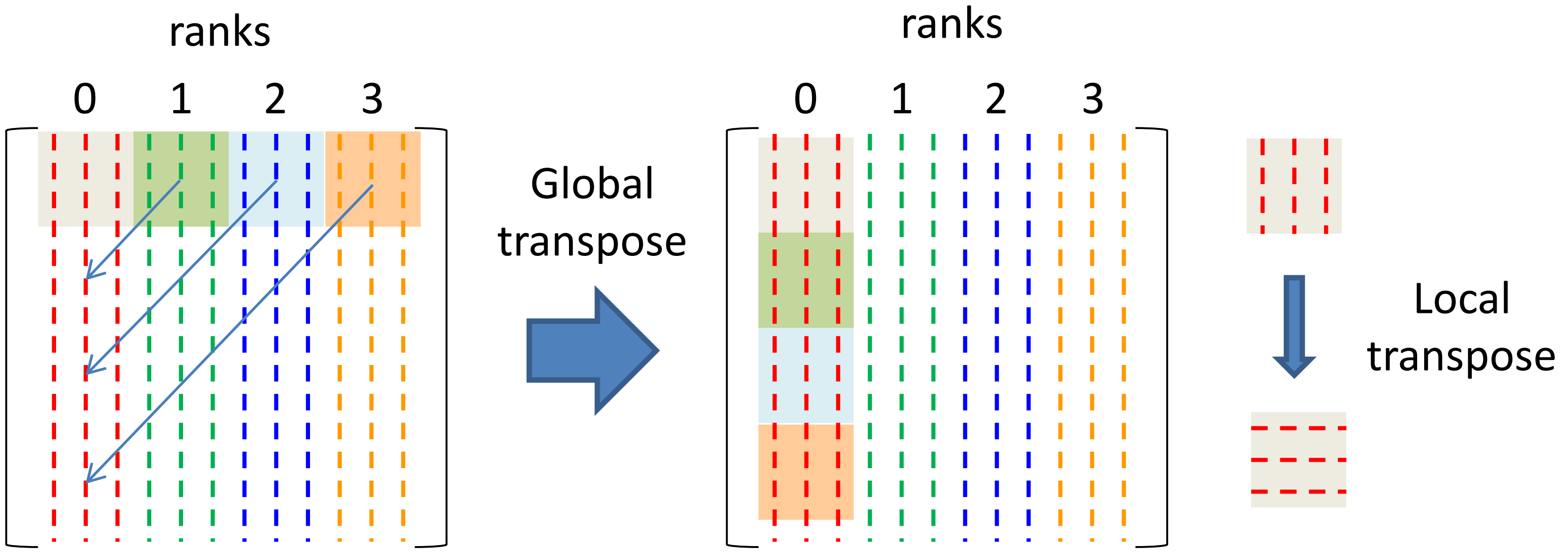
**embarrassingly parallel*

Dense matrix Transpose

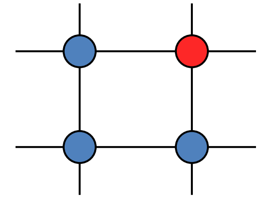
What: $A = B^T$, both A and B distributed identically, and by whole columns, using column- major storage format

Why: Stride 1 on read, large stride on write (TLB, cache misses). All to all communication. Common operation in HPC (FFTs, numerical linear algebra)

How: Implemented in MPI, MPI+OpenMP, MPI+MPI3-shared memory, Charm++ (tiled local transpose to reduce misses)



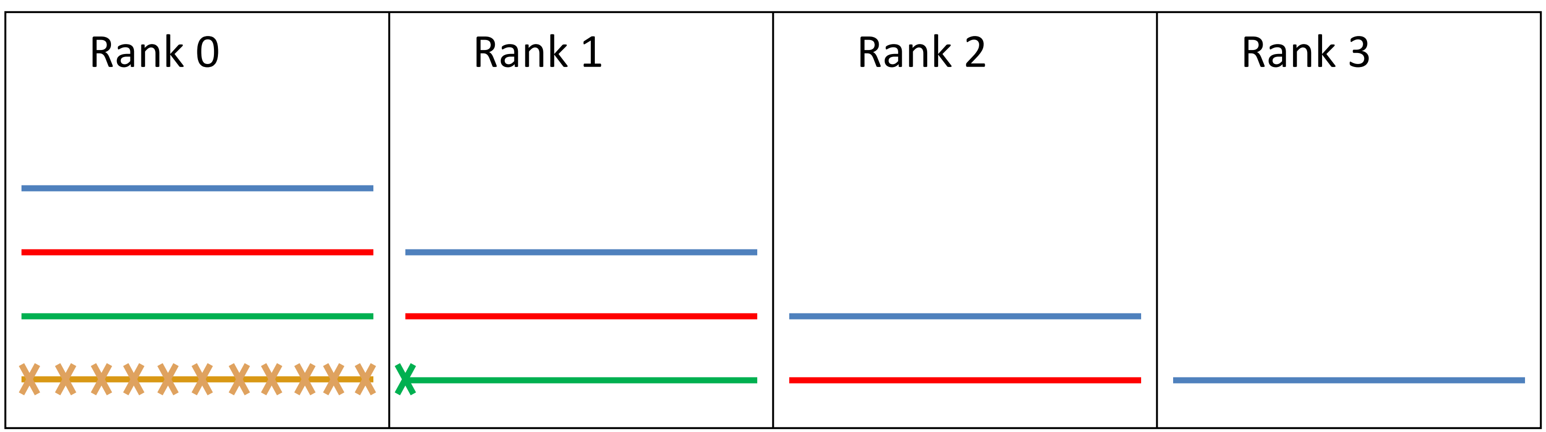
Point to point synchronization (Synch_p2p)



What: $A(i,j) = A(i-1,j) + A(i,j-1) - A(i-1,j-1)$
Pipelined solution of problem with non-trivial 2-way dependencies

Why: Threads/ranks forced to synchronize pairwise frequently during each iteration. Stresses fine grain communication/synchronization.

How: Implemented in MPI, MPI+OpenMP, MPI+MPI3-shared memory, Charm++, Grappa

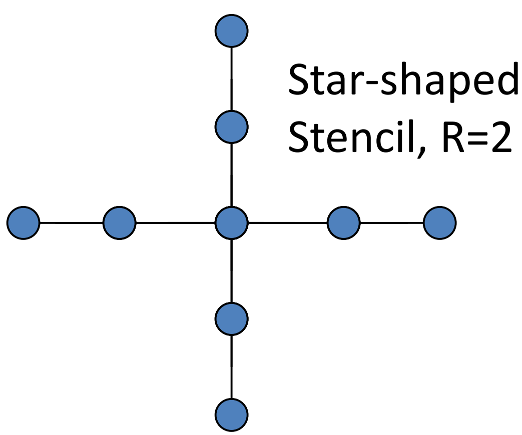


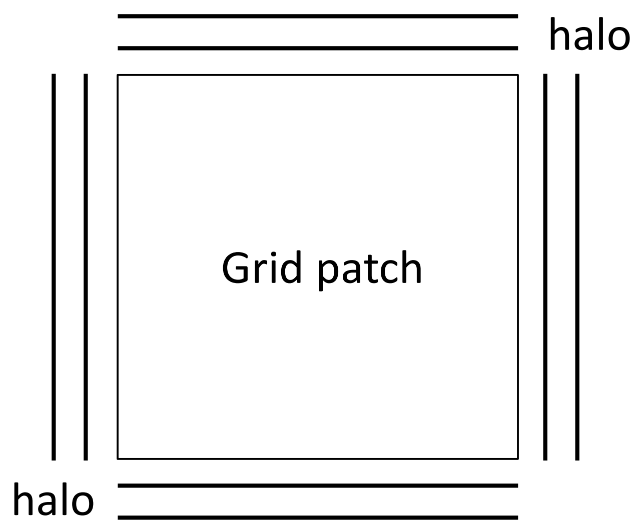
Stencil computation

What: For all points in 2D grid, compute a = S(b), S is star-shaped stencil operator (radius R), a and b are scalar grid variables (2D arrays)

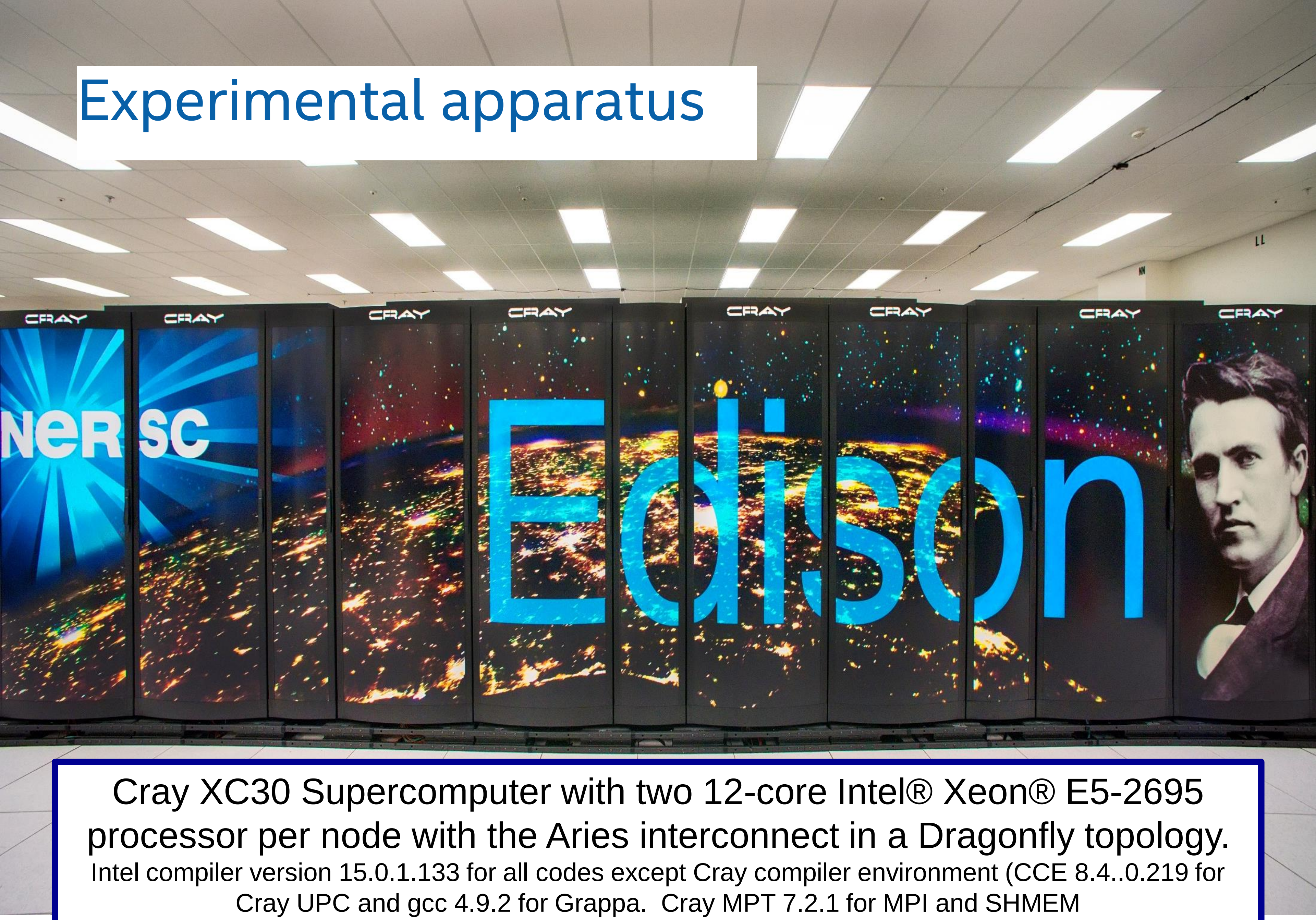
Why: Data parallel; point-to-point bulk communication; common operation in HPC

How: Implemented in MPI, OpenMP, MPI+MPI3-SHM, MPI+OpenMP, Charm++, Grappa; use 2D decomposition, collect halo values at start of each iteration



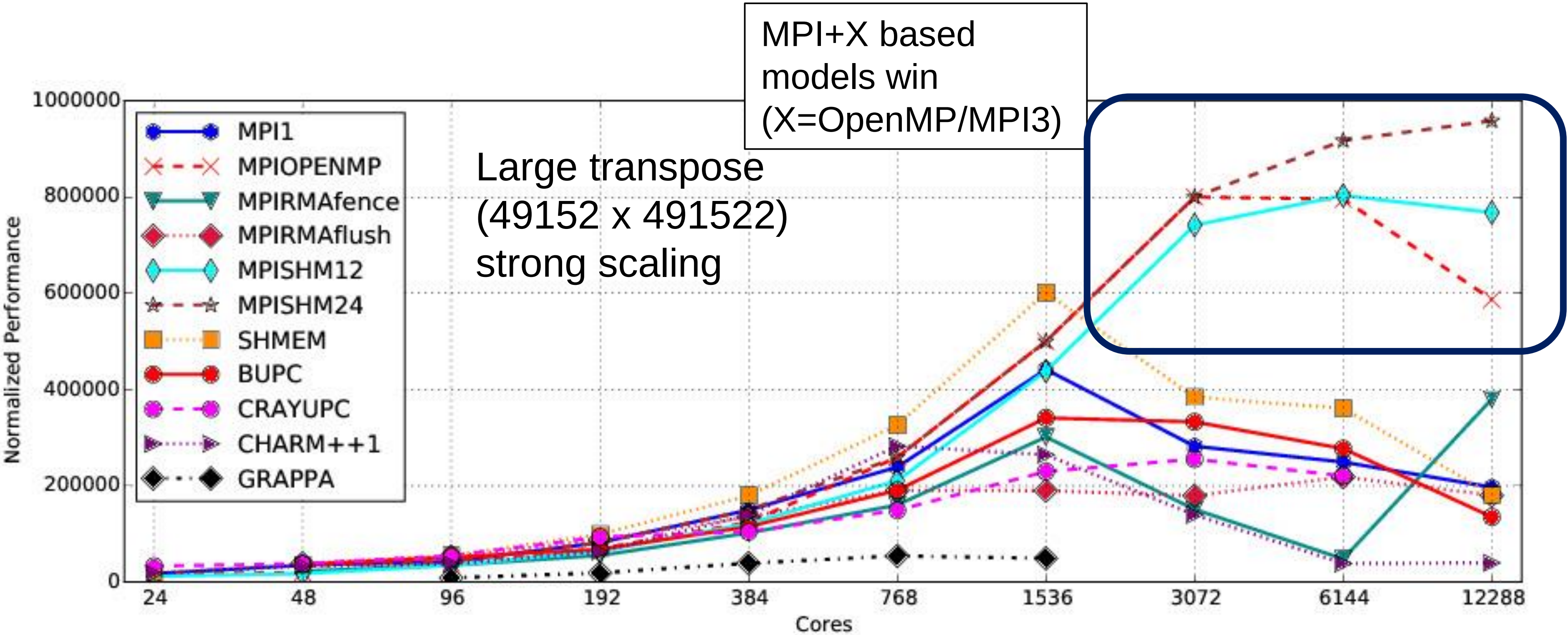


Experimental apparatus



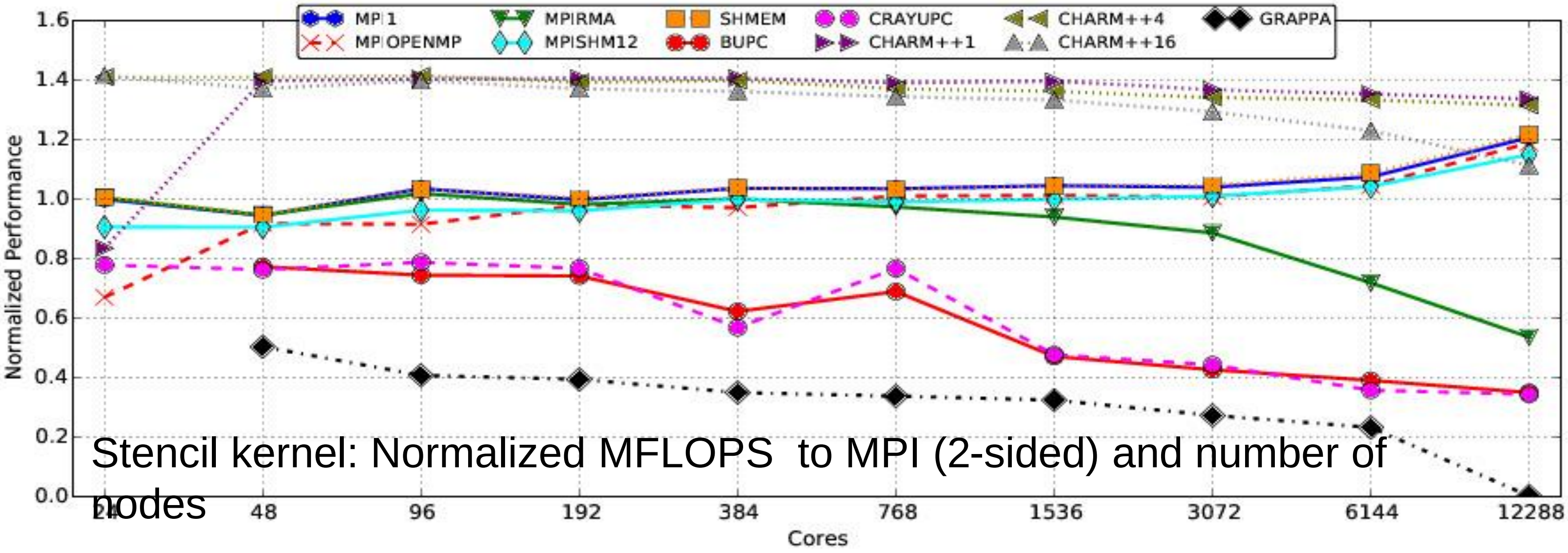
Cray XC30 Supercomputer with two 12-core Intel® Xeon® E5-2695 processor per node with the Aries interconnect in a Dragonfly topology.
Intel compiler version 15.0.1.133 for all codes except Cray compiler environment (CCE 8.4..0.219 for Cray UPC and gcc 4.9.2 for Grappa. Cray MPT 7.2.1 for MPI and SHMEM

Transpose Parallel Research Kernel (PRK)



- MPI-OpenMP, MPI-SHMEM do very well since this coarse grained kernel is ideally suited to their SPMD pattern
- Grappa generates large numbers of small messages as the core-count increases .. .hence its performance is poor.

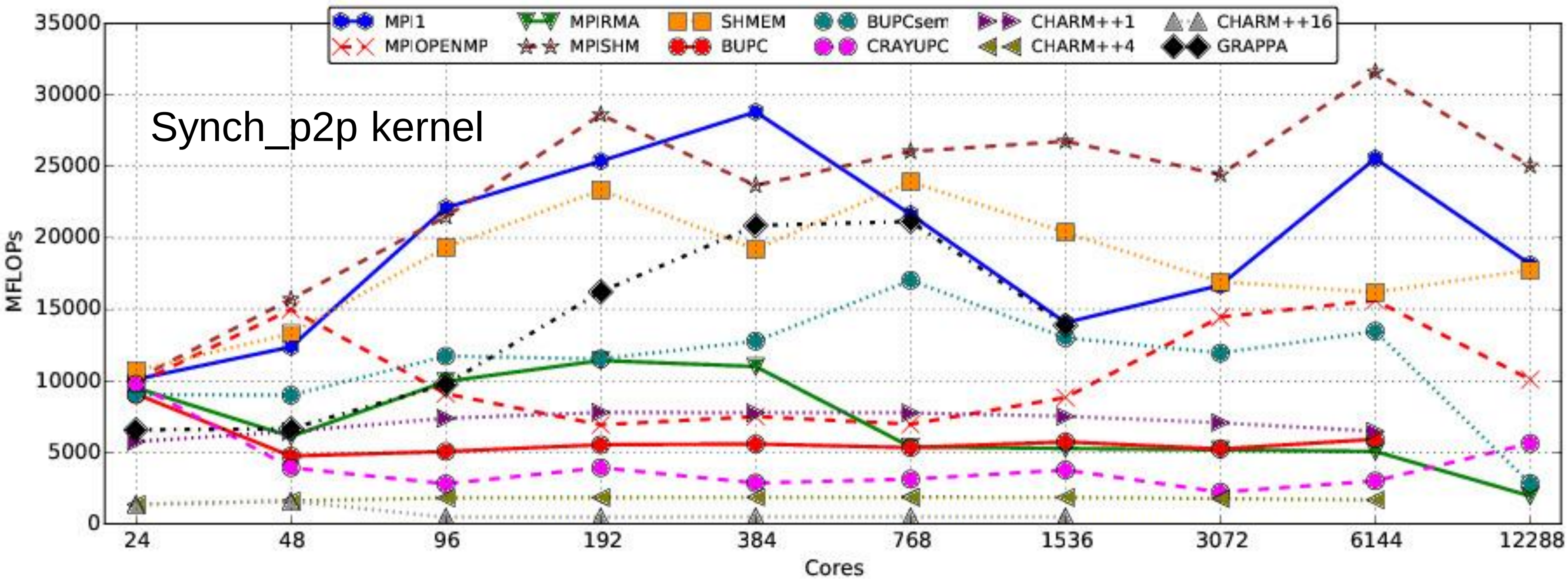
Stencil Parallel Research Kernel (PRK)



Stencil kernel: Normalized MFLOPS to MPI (2-sided) and number of nodes

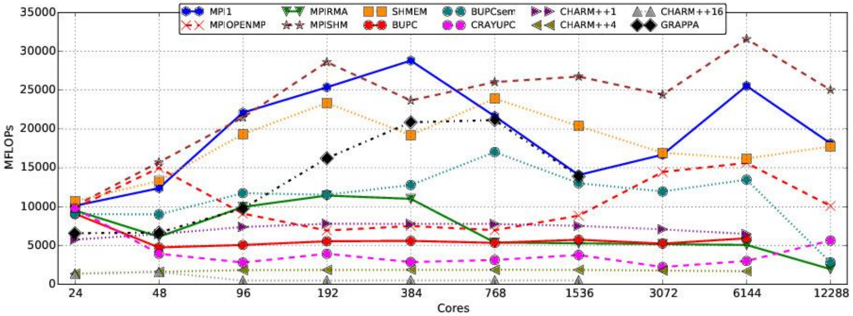
- Runtimes that depend on barriers (MPIRMA, BUP, CrayUPC) do not do well.
- Runtimes that use pair wise sync (Charm++, Grapa, SHMEM, MPI1) scale well.
- Charm++ handles this example particularly well (numbers 1, 4, and 16 are the degrees of over decomposition)

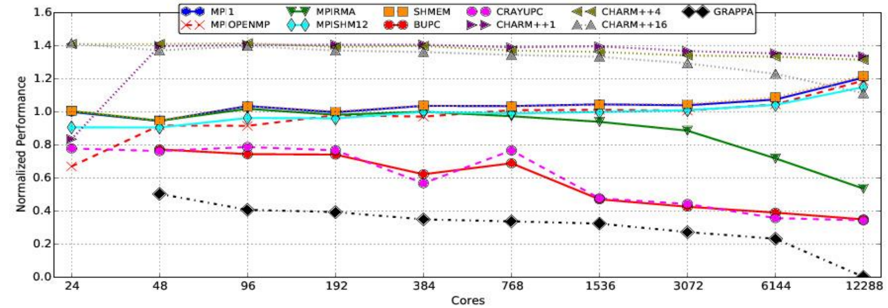
Synch_p2p Parallel Research Kernel (PRK)

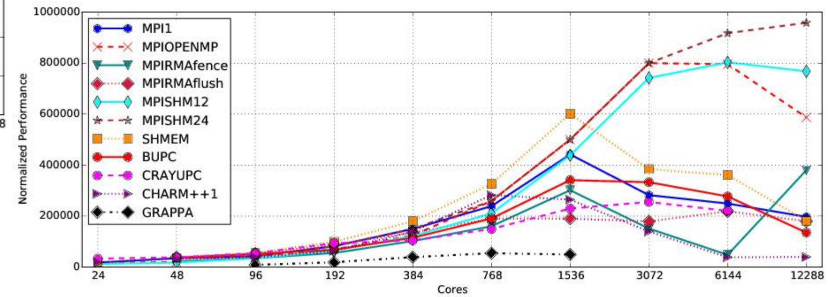


- Performance dominated by (1) frequent pair-wise synchronization and (2) global sync after each iteration.
- MPIRMA and both versions of UPC used barriers after each iteration … this really hurt their pefformance.
- MPI1, SHMEM, and MPISHM have very similar perf.
- This workload is poorly suited to the dynamic execution profile charm++ was designed for.

How much should we emphasize performance







- Modern Assynchronous Multitasking (AMT) Systems are immature compared to MPI and OpenMP.
  - It’s difficult to separate pathological problems with the model from low immature implementations of models.
- What really matters is programmability … how effective are these programming models for application programmers?

Programmer Experience notes: Legion

- No language spec, no documentation, no user-manual.
  - Only doxygen based on source code documentation
  - Hard to identify, discover some of the runtime API
- Need to learn too many new terms, structures specific to Legion
  - Logical regions, physical regions, fields, index spaces etc.
- Physical Region accessors are not intuitive, no direct array based indexing, e.g. value = acc_a[i]
- Due to the live development, sometimes hard to decide whether you hit a Legion runtime bug or not
- Collectives are not user friendly as advertised
  - IndexSpace tasks should have a direct reduction operation on the calculated Future value but API requires a lot of verbosity to implement this in the code
  - Again finding the correct API, best method is cumbersome
  - Still working on resolving some ambiguities on this

Programmer Experience notes: Charm++

- Not having a Charm++ compiler greatly hurts productivity.
  - Many programmer errors result in *auto-generated header files* that trigger compiler errors.
  - Interface files (.ci) have special syntax that looks like C++, but isn't.
  - Interface file parser needs to be expanded each time an unexpected construct is encountered—result of no interface file language specification.
  - Collectives in interface file must have different names, even if they do identical things. Consequence: reusing code is harder than it should be.
  - Global variables must be declared in .C file **and** in interface file. Failure to declare these as *readonly* in interface file leads to runtime errors, not compiler errors.
  - Having to declare code as *structured dagger* code is incomprehensible to the programmer. It is just Charm++.
  - Making the programmer responsible for functions **and** their proxies (must maintain two sets of names) is awkward at best, and invites errors.
- Building Charm++ on top of MPI and/or using non-gnu tools not obvious.

Programmer Experience notes: Grappa

- No user manual.
- No “threadprivate” variables, except through global variables.
- Support for global variables weak/buggy.
- Only two types of synchronization: barrier, and full-empty bits; leads to inefficient code.
- Full-empty bits implementation incomplete.
- Compile and link flags not statically determined; must be copied from special location after Grappa build.
- Building Grappa with Cmake not obvious.
- Having to support external repos or hack around them cumbersome.

Programmer Experience notes: OCR

- The specification is GREAT!!!
- For something as old as OCR, the implementation on scalable systems is way behind the competition.
- It is not programmer friendly … yes it's a runtime and not designed for application programmers, but still, it shouldn't make "the easy things so hard"

Advice to the AMT community

- AMT is like communism …
  - Its a great idea but we just don’t have any good implementations to reference.
- Doxygen is NOT useful documentation for applications programmers.
  - Write a specification ... It will be good for you
  - Write a programmers reference manual
  - Create an official examples suite to help people learn.
- Don‘t fight MPI on it‘s own turf.   The advantage of AMT that is hard for MPI+OpenMP to match is resilience. So why is it that only Charm++ has a mature resilience implementation?
- Remember if you compete on MPI‘s turf, (paraphrasing the old joke about hikers and bears ..)
  - *MPI does not need to outrun the bear (exascale requirements), just the other hikers (disruptive runtimes), to be a winner*

AMT: Asynchronous multitasking

Ease of use

- Once you climb the MPI learning curve, it's a natural and easy model to work with:
  - The people I hear complaining the most about MPI are the ones who don't use it … often AMT researchers.
  - MPI programmers like MPI!!! It runs everywhere and is easy to debug.
- Key change I'd like to see in the AMT community:
  - Stop making the things that are easy in MPI/OpenMP so hard:
    - Collectives
    - Launching large SPMD programs across a scalable system
  - Legacy code is far more important than many people think. If your solution to my programmability program begins with the phrase .. "Recode everything in my new language", I'll stop listening to you.
- A good AMT model must make it easy for me to program "in the large" with algorithms I care about, not the ones that nicely match what your programming model expects.

2 pathways to Exascale Runtime Research

Evolutionary
(e.g. MPI+X)

Revolutionary
(e.g. OCR)

Based on our experiences with AMT systems and understanding of you application programmers work, we are choosing a path to exascale based on MPI.

Systemic Exascale Challenges

- System Utilization
- Asynchrony
- Data movement cost
- Load Imbalance
- Fault Tolerance
- Scalability

MPI for Darwinists:

An extended MPI that coherently integrates disruptive new features

1. Fault Tolerance in MPI
   - Local checkpoint/restart for task rollback, ability to serialize and migrate ranks. (collaborators: Fenix and UFLM)
2. Transparent Over decomposition in MPI:
   - Ranks-as-threads integrated into the MPI runtime. (collaborators: Fine Grained MPI Group)
3. Infrastructure to support load balancing
   - Rank migration (collaborator: AMPI)
4. Investigate new, light-weight load balancing schemes (on and between nodes):
   - MPI+MPI: Rank stealing
   - MPI+X: OpenMP 4+ (tasks, target), TBB

But this is all just anecdotal evidence. Is that really enough?

- Programmability is too important to leave to anecdote.
- We need a systematic framework to access programmability.
  - HPCS program made a noble attempt to create such a framework:
    - Measured actual programmer productivity
    - Found data showing that (1) once a project chooses a language, they won't change; (2) OpenMP is NOT harder to debug than MPI; (3) programmers care about portability, performance and the ability to make incremental changes more than elegance.
    - Constructed formal models including some based on economic utility theory.
  - The HPCS program, however, had no lasting impact on programmability. Why? My untested hypothesis
    - They didn't engage cognitive psychologists in their studies of programmers and hence didn't not come up with models based on how human's think about programming … it was hard to turn the HPCS results into action

Psychology of Programming in one slide

- Human reasoning is model based … Programming is a process of successive refinement of a problem over a hierarchy of models.[1]
- Programmers use an informal, internal notation based on the problem, mathematics, programmer experience, etc.
  - Within a class of programming languages, the solution generated is only weakly dependent on the language.[2] [3]

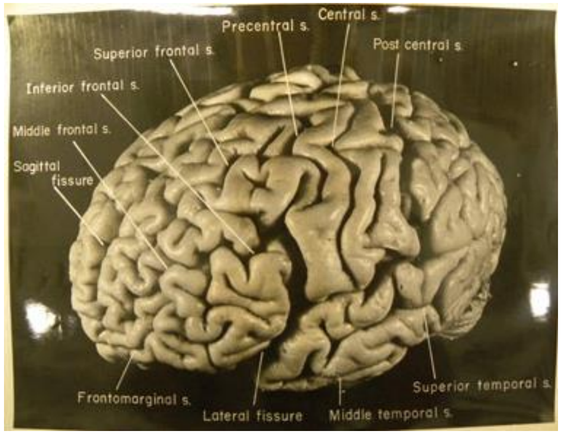


Image source: Einstein's brain from https://scim.ag/Einsteinbrain. Collected on 12/4/2012

- Programmers think about code in chunks or "plans". [4]
- Opportunistic Refinement: [5]
  - Progress is made at multiple levels of abstraction with effort focused on the most productive level. It's not top-down or bottom-up …. We bounce!

[1] R. Brooks, "Towards a theory of the comprehension of computer programs", Int. J. of Man-Machine Studies, vol. 18, pp. 543-554, 1983.

[2] S. P. Robertson and C Yu, "Common cognitive representations of program code across tasks and languages", int. J. Man-machine Studies, vol. 33, pp. 343-360, 1990.

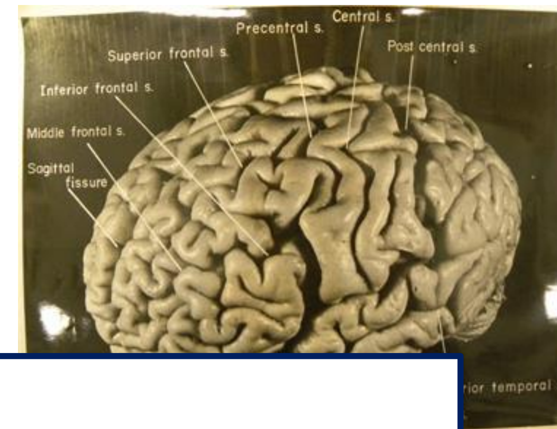
[3] M. Petre and R.L. Winder, "Issues governing the suitability of programming languages for programming tasks. "People and Computers IV: Proceedings of HCI-88, Cambridge University Press, 1988.

[4] R.S. Rist, "Plans in programming: definition, demonstration and development" in E. Soloway and S. Iyengar (Eds.), Empirical Studies of Programmers, Norweed, NF, Ablex, 1986.

[5] M. Petre, "Expert Programmers and Programming Languages", in [Hoc90], p. 103, 1990.

Psychology of Programming in one slide

- Human reasoning is model based … Programming is a process of successive refinement of a problem over a hierarchy of models.[1]
- Programmers use an informal, internal notation based on the problem, mathematics, programmer experience, etc.
    - Within a class of programming languages, the
      so
      th
- Progr
- Oppo
    - – n
      nce!



12/4/2012

This implies that we want programming models that:

- Make each layer in the hierarchy of models explicitly visible.
- Abstract lower layers but don’t hide them.
- Programmers think in “plans” so make them explicitly visible.
- Programmers must be able to control the level of abstraction they work within … and they need to bounce

[1] R. Brooks, "T

[2] S. P. Roberts 33, pp. 343-360, 1990.

[3] M. Petre and R.L. Winder, "Issues governing the suitability of programming languages for programming tasks. "People and Computers IV: Proceedings of HCI-88, Cambridge University Press, 1988.

[4] R.S. Rist, "Plans in programming: definition, demonstration and development" in E. Soloway and S. Iyengar (Eds.), Empirical Studies of Programmers, Norweed, NF, Ablex, 1986.

[5] M. Petre, "Expert Programmers and Programming Languages", in [Hoc90], p. 103, 1990.

Conclusions

- Programming model developers
  - Innovate and publish, but your output should be to improve/evolve existing standards.
  - Exception … watch for hardware inflection points. That's the only time new models flourish
  - Sociology is as important as technology … choice overload and "worse is best" are real.
  - We need to develop a formalism for programmability grounded in the psychology of programming.
- Some key Results with the parallel research kernels
  - Flat MPI often performs best
  - MPISHM usually close, sometimes much better (Transpose); gives greatest flexibility in tuning granularity
  - MPI+OpenMP in canonical form never competitive
  - Charm++ competitive only for medium grain workloads; little/no benefit seen of over-decomposition
  - Legion … we've been working with Legion for almost two years with little to show. Brutal learning curve of a moving target