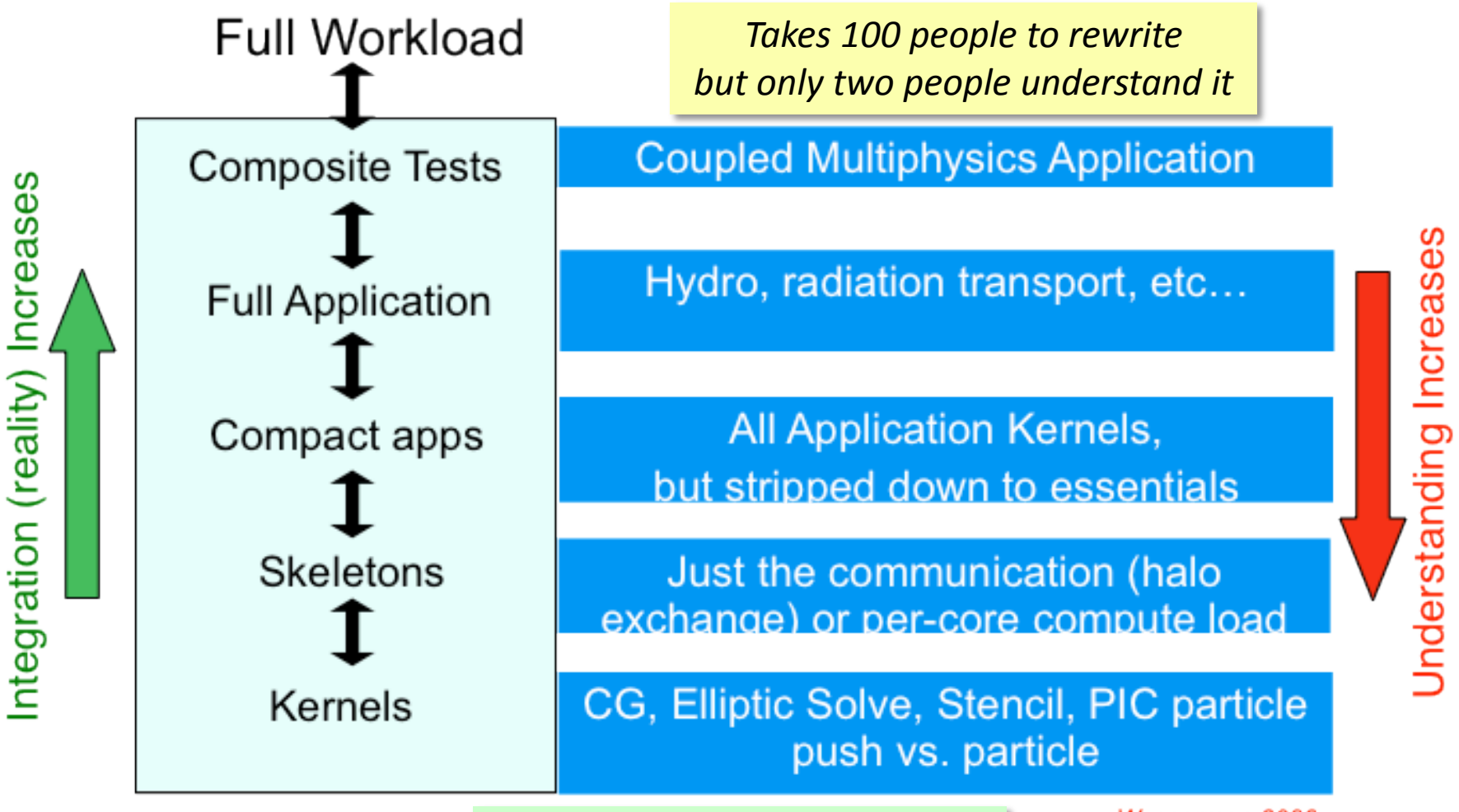
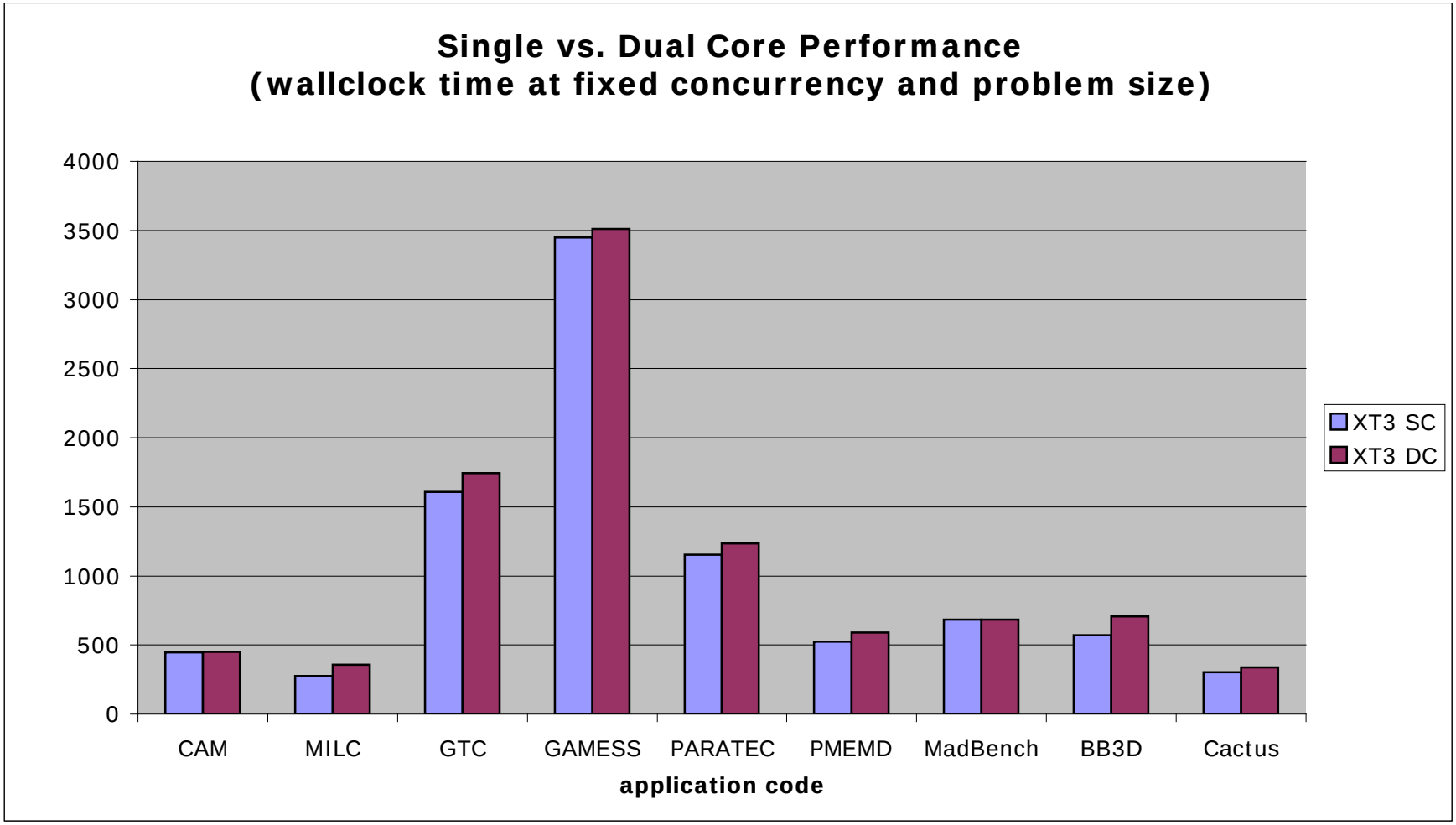
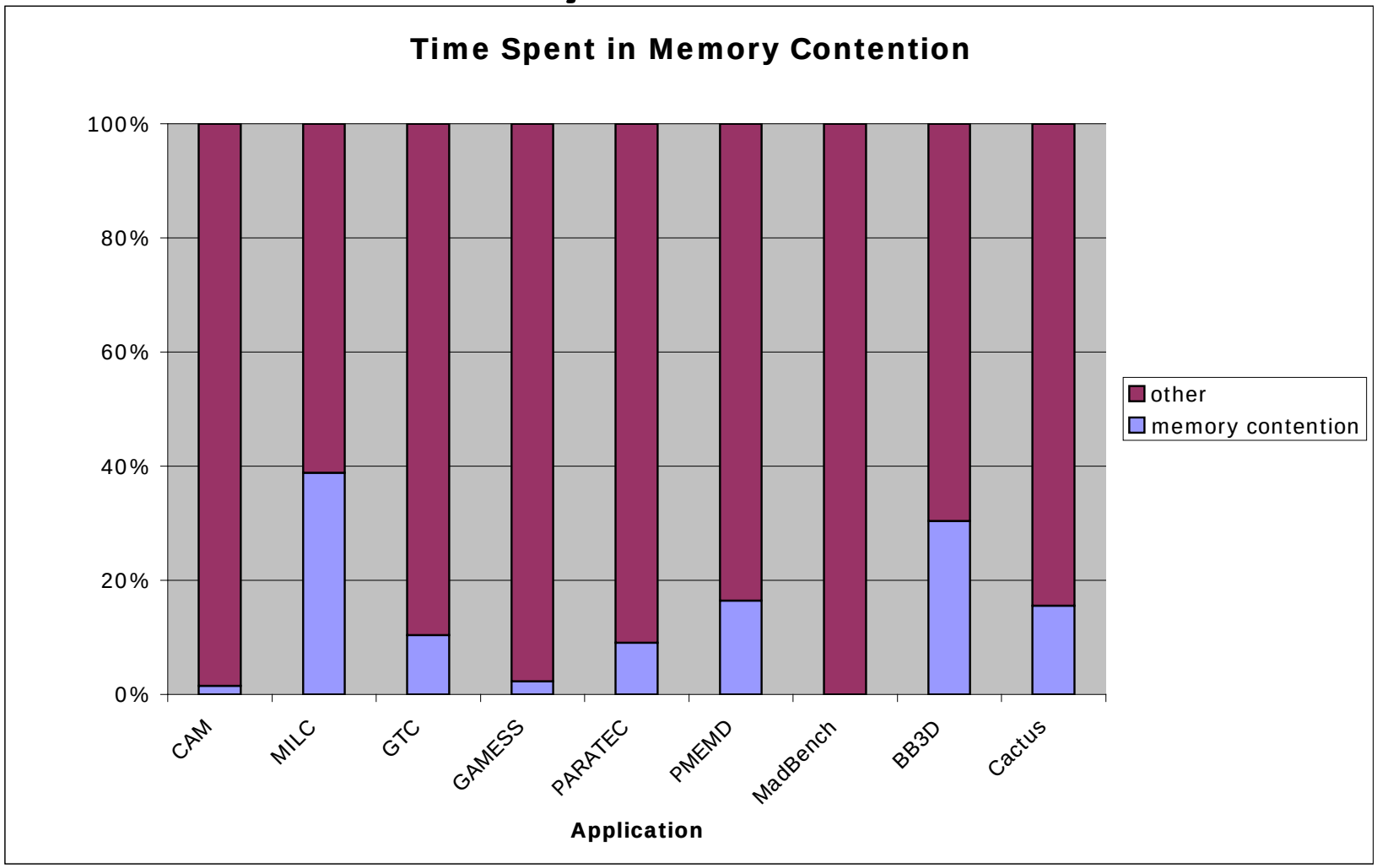
Hierarchy of Reduced Applications



NERSC SSP Applications

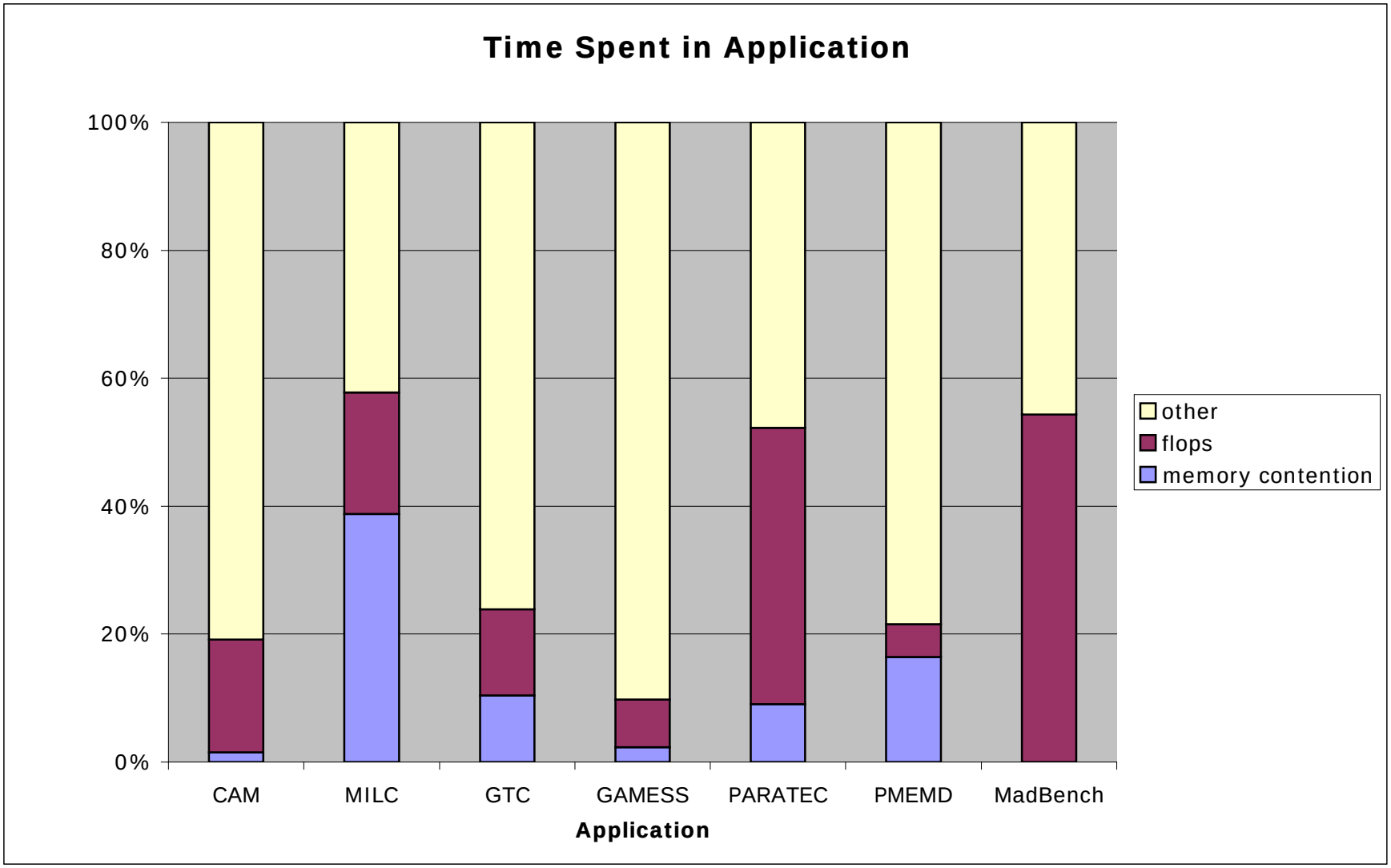


Memory Contention



"Other" may include *anything* that isn't memory bandwidth)
(eg. latency stalls, Integer or FP arithmetic, I/O.)

Contribution of FLOPs to exec time for NERSC SSP apps



The Resurgence of Functional Programming and Dataflow

John Shalf

NERSC Advanced Technology Group
Lawrence Berkeley National Laboratory

ASCR Workshop on Exascale Programming Challenges, Marina Del Rey
July 27, 2011

U.S. DEPARTMENT OF ENERGY | Office of Science

NERSC National Energy Research Scientific Computing Center

BERKELEY LAB Lawrence Berkeley National Laboratory

Back to the Future:

Functional Languages and Nouveau Dataflow to Tackle Asynchrony and Parallelism

U.S. DEPARTMENT OF ENERGY | Office of Science

NERSC National Energy Research Scientific Computing Center

BERKELEY LAB Lawrence Berkeley National Laboratory

Dataflow 2.0:

Will it work this time?

U.S. DEPARTMENT OF ENERGY | Office of Science

NERSC National Energy Research Scientific Computing Center

BERKELEY LAB Lawrence Berkeley National Laboratory

DAG! Its Dataflow Again!

Overview

- Discuss the merits of functional languages for simplifying parallelism in many dimensions
- Functional languages get rediscovered by programming community every 10-15 years or so because of elegant expression of parallel constructs (coincide with power crises)
- However, past forays have met strong resistance from scientific applications community
- Will they take root this time around
  - What was good about them? What was bad about them? *(know benefits and know the baggage!)*
  - Are there any lessons learned? *(we have at least 30 years+ experience, and lots of buried bones to dig up)*

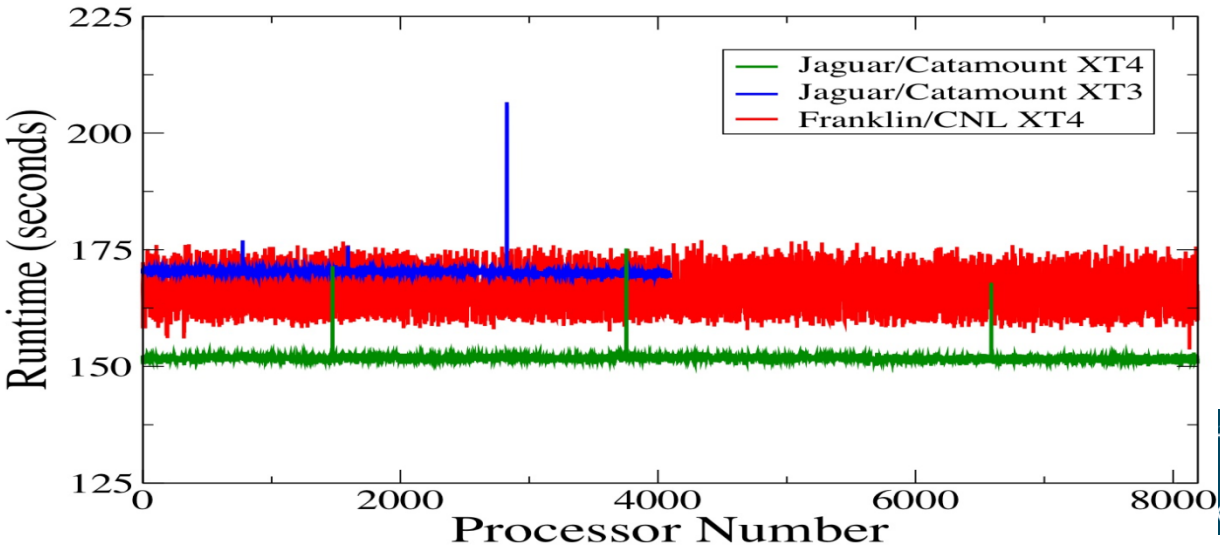
Where are We Today

- **Addicted to Bulk-Sync/SPMD Programming Model**
  - Low Cognitive Load
  - Everyone does the same thing at <appoximately> the same time
  - Data and control hazards are isolated to epochs of code execution *(not all possible interleavings of threads described in "The Problem with Threads")*

- **SPMD Models Have Demanding Requirements for Hardware/Software Ecosystem**
  - Homogeneous execution rates
    - Homogeneous performance per core (control OS "Noise" for example)
    - Homogeneous work per core (must code-your-own load balancing for adaptive algorithms)
  - Fast sync/collective operations (BG collective network)
    - Has similarity to instruction bcast for SIMD
  - Exhausting Sources of parallelism through domain decomposition
  - Gravitate towards bulk-sync communication
    - To make it easier to reason about control flow/messaging hazards
    - Creates episodic floods of interconnect traffic
    - *try to mitigate by getting overlap*

Trouble on the Horizon

- **New sources of inhomogeneity in hardware**
  - Sparing redundant resources to tolerate hard errors creates inhomogeneous communication characteristics
  - constrained interconnect topologies (graph embedding for comm topology)
  - Inhomogeneity in process technology leads to non-uniform clock rates
  - Thermal Throttling (Intel Sandybridge)
  - Hardware fault recovery to tolerate transient errors (software recovery mechanisms will make it even worse)



Trouble on the Horizon

- **Algorithm/Application requirements**
  - ***Adaptive Algorithms/AMR:*** Fastest, most energy efficient FLOP is the one you don't execute
  - ***Irregular structure:*** Irregular mesh, Sparse matrix, and MD computations have irregular work and dependency patterns (DAG scheduling & Curt)
  - ***Irregular work:*** Subcycling for ODEs for combustion chemistry or to squeeze out residual error for fluids probs creates inhomogeneity for regular structures
  - ***Domain Decomp:*** Exhausted parallelism through domain decomposition motivates move towards functional decomposition (climate coupler/Mike Heroux)
  - ***Software Engineering:*** Separation of concerns for frameworks & libraries

- **New Memory Hierarchies and structures**
  - Non-coherent Global Address Space or Cache Coherence with relaxed consistency Model??? *when do I sync?*
  - **Disjoint memories**: marshalling/unmarshalling data for accelerators & scratchpads

The Fix is In

- **Looking for a post-SPMD programming model**
  - Much hope placed on asynchrony and async execution models to overcome these rapidly emerging problems *(crossing fingers… and waving arms)*
  - Hard to manage this with imperative pmodel (overspecifies)
  - This implies functional semantics (or something like it) to make use of async models tractable
  - Be careful what you ask for (hidden baggage)

- **Imperative programming languages make it hard to move to post-SMPD / asynchronous execution models**
  - Current languages (C/Fortran) over-specify implementation / solution
    - hard to prove that something can be executed in anything other than non-program order
  - Unbounded ability to modify global state makes dependence analysis difficult
  - Unbounded ability to modify global state makes it hard to determine what state is associated with unit of computation
    - Hard to automagically copy to a disjoint memory: accelerators, scratchpads
    - Hard to automagically determine minimal state to preserve for fault tolerance (recovery from transient errors)
    - Can't figure out what state to migrate with computation to fix load imbalances
  - Auto-parallelization is hampered by inability to analyze code to look for alternative schedule or structure

Functional Semantics

- **Requirements:** *express computation declaratively*
  - Stateless
  - No side-effects
  - Only operate on data you were handed

- **Benefits of "Isolation"**
  - Data dependence becomes statically analyzable
  - Exposes implicit parallelism (DAG as constraint and runtime has a lot of freedom to control schedule)
  - Trivial data migration or task migration (containment)
    - Local stores, accelerators and other disjoint memories are not a problem
  - Know where data is needed OR when it is needed (but getting both is hard)

Relation to Communication Primitives

- **Functional Semantics just provide guarantees that enable you to use *SOME* primitives more productively, but does not imply a specific set of primitives or an implementation**
  - **Scheduling**: Makes schedule constraints obvious (ID bad schedule easy)
    - But does not make scheduling itself any easier (still a hard problem)
    - And OS makes it more complicated again because lack of control
    - Need more sophisticated runtimes (but OS is in the way again)
  - **Synchronization**: makes it clearer when to synchronize and avoid over-synchronizing
    - but sync primitives get costly if granularity too small
  - **Communication**: Makes clear when & what to communicate
  - **Partitioning**: Doesn't directly address this (Sequoia and CILK use recursion + codelets for managing partitioning)
  - **Placement**: Can get good temporal placement, makes data movement easier, but doesn't really solve broader partitioning problem
  - **Migration:** Placement+Communication is made trivial (totally awesome… easy to use scratchpad) But does not solve ephemeral load imbalance prob.

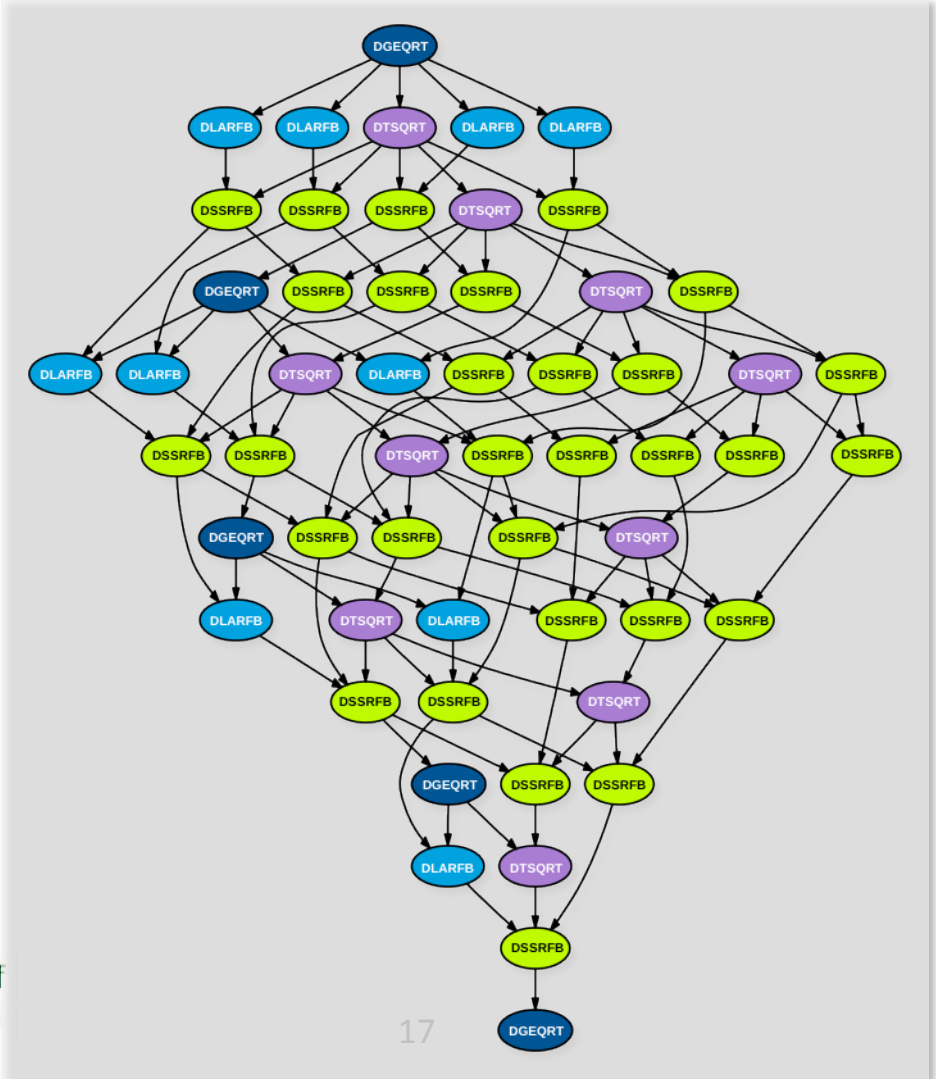
Options to Express Functional Semantics

1. ***Exceedingly Careful Programming***
   - Good software engineering obeys functional semantics at coarse level (Rusty: Science of FW)
   - Enforced by convention, good software engineering, and/or stuff breaking if you fail to obey
   - Examples: Mike Heroux/Trillinos, PLASMA/MAGMA DAG Scheduling, Cactus, SIERRA, Curt's talk, CnC and TBB assists implementations
2. **Directives, Pragmas and other annotations** *(smuggle in critical missing information)*
   - Integrate with existing C/Fortran code
   - Does not promote parallel thinking *(see Guy Steele's rants..)*
   - Incorrect use will make functioning serial code an "error"
   - Verbosity *(see Bob's presentation)*
   - Pervasiveness *(have to touch every "important" loop in your code)*
   - Examples: OpenMP 3.0, HMPP, and various other Accelerator directives
3. **Type systems to express locality and scope of reference**
   - Can make locality inference easier (Example: PGAS shared array types)
   - Can make dependency analysis easier (Example: SSA, Ct TVECs and Intel ArBB)
   - Special meaning for types in call-list (example: CILK and Berkeley IVY typesys)
4. **Functional Languages and full functional semantics**
   - Statically analyzable, exposes implicit parallelism
   - Provides scheduling freedom (can know when its legal or illegal to schedule things)
   - But makes familiar conveniences such as linked-lists damn near impossible

Nouveau Dataflow

DAGs for Irregular Structures

- UT/Dongarra: MAGMA/PLASMA for sparse matrices



Nouveau Dataflow: Intel Ct

- Channeling NESL and SISAL through C++
- Uses template classes to implement single-assignment arrays
  - TVEC implements Single Assignment Arrays using Templates
    - TVEC<float64> DoubleVec; DoubleVec = Vec1*Vec2
    - Assignment of value creates “new” version of array rather than modifying value in the array (dataflow “futures” for arrays)
  - Ct Lambdas on TVECs express element-wise parallelism

- Underlying runtime system manages parallelism if you make use TVECs and their resulting semantic guarantees
  - Featherweight Threads: “Futures”
  - Support for Dataflow “patterns” (reduced sched overhead) and/or work-stealing job scheduler that obeys dataflow constraints
  - But if you don’t use Ct runtime, its still functionally correct

CILK *Leiserson et al.*

- **Th** **tional**
  lan **tate)**
  - –
  - –
  - – ocality
- **Se**
  - – to
    managing variable domain decomposition for different memory hierarchies
  - – Difficulty in that processes at same level of hierarchy can only communicate through parents

Fibonacci

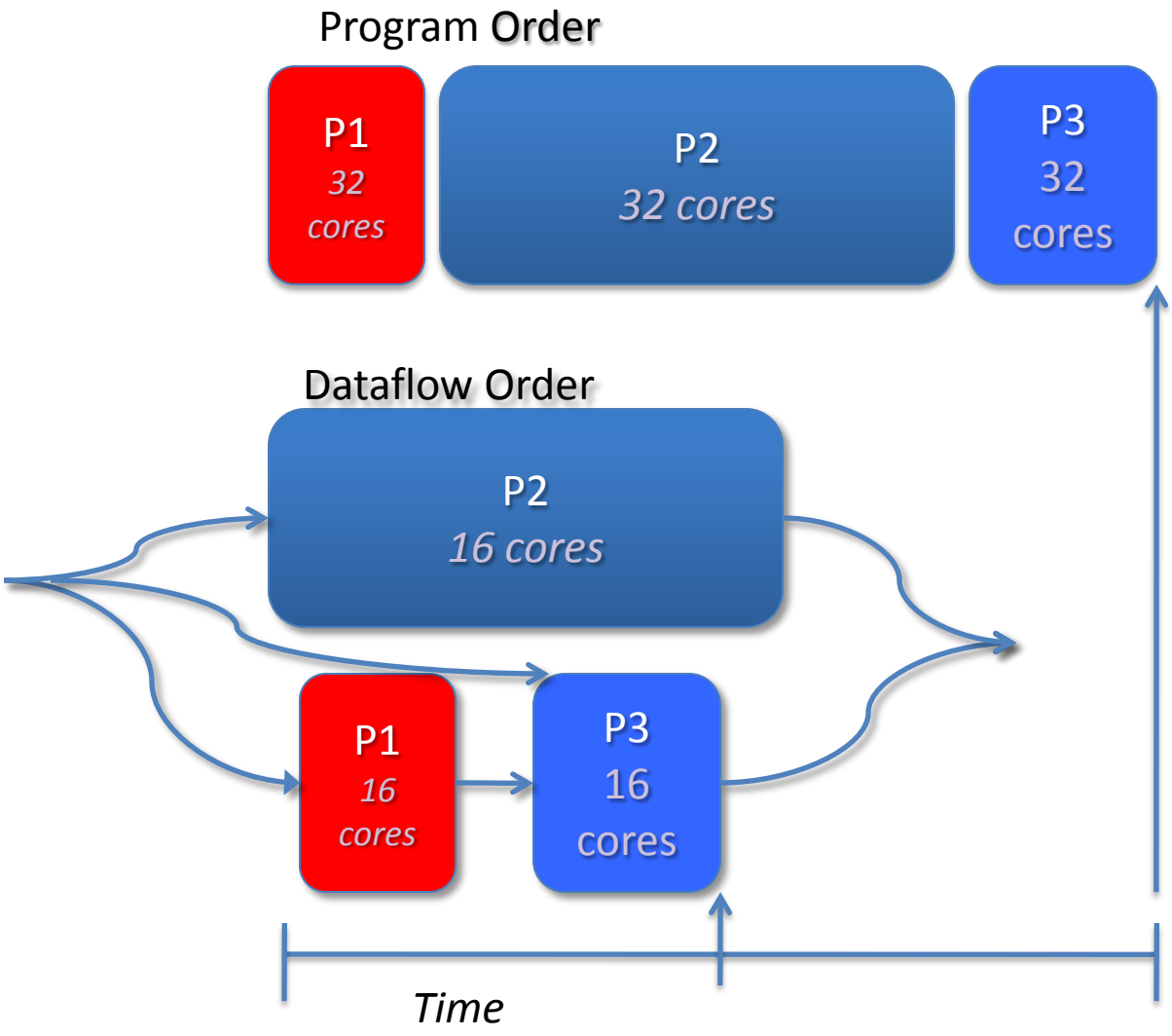
```
int fib (int n) {
if (n<2) return (n);
  else {
    int x,y;
    x = fib(n-1);
    y = fib(n-2);
    return (x+y);
  }
}
```

C elision

Cilk code

```
cilk int fib (int n) {
  if (n<2) return (n);
  else {
    int x,y;
    x = spawn fib(n-1);
    y = spawn fib(n-2);
    sync;
    return (x+y);
  }
}
```

Functional Partitioning to Reduce Pressure on Domain Decomposition



Examples using TBB (functionally complete, but overheads high)

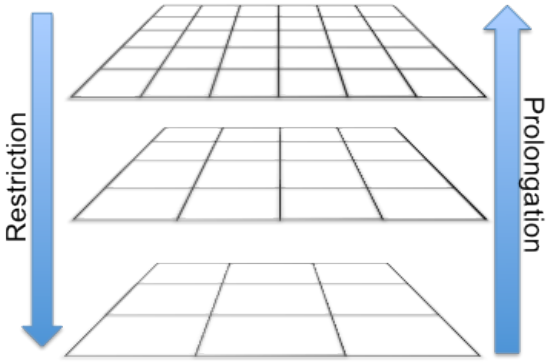
Hand-rolled impls. Libraries to formalize

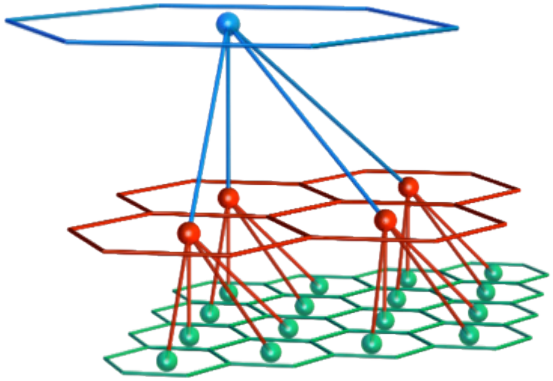
Schedule independent physics To Execute Concurrently

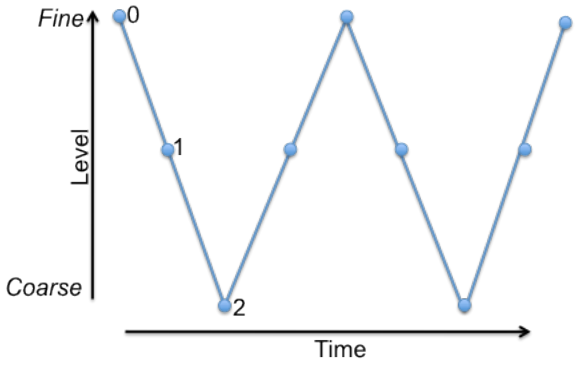
This is hard to do without functional semantics

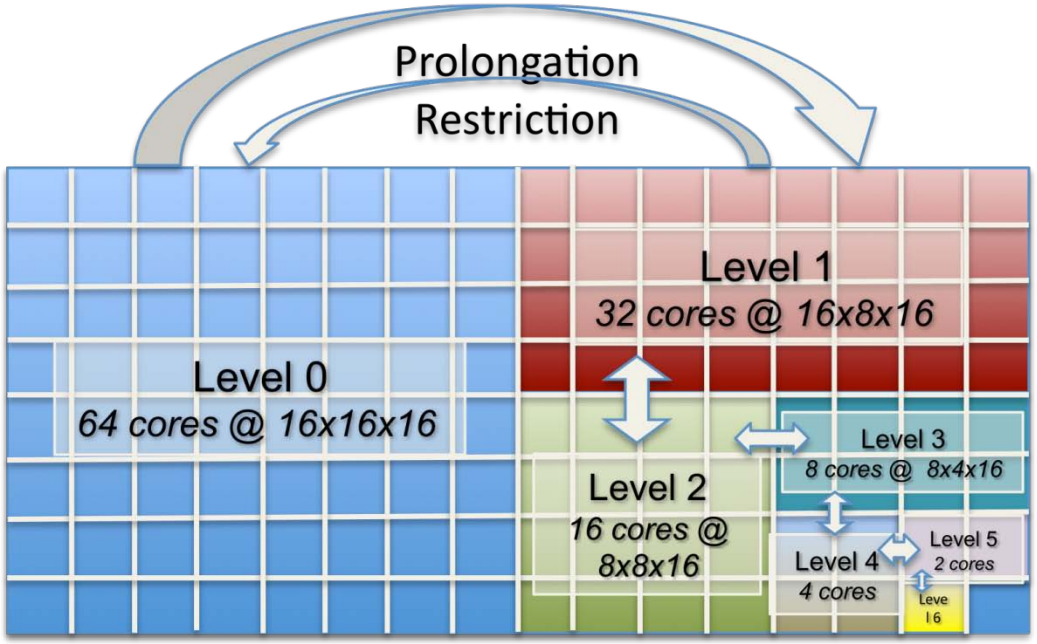
Example Multigrid Elliptic Solver

(7 levels on-chip using feed-forward pipelining)









With lightweight cores, we can no longer domain-decompose problems and get a speed-up.

Use MIP-Mapping technique
To pack multiple MG levels onto the chip and pipeline accesses and reduce off-chip accesses.

Would benefit from interprocessor message queues (default is mutex-protected queues like TBB)

Functional Style for Domain-Specific Frameworks and Embedded DSL's

- **Frameworks and domain-specific languages**
  - enforce coding conventions for big software teams
  - Encapsulate a domain-specific "idiom for parallelism"
  - Create familiar semantics for domain experts (more productive)
  - *Clear separation of concerns (separate implementation from specification)*
- **Common design principles for frameworks from SIAM CSE07 and DARPA Ogden frameworks meeting**
  - Give up main(): *schedule controlled by framework*
  - Stateless: *Plug-ins only operate on state passed-in when invoked*
  - Bounded (or well-understood) side-effects: *Plug-ins promise to restrict memory touched to that passed to it (same as CILK)*

- *By gosh, these are attributes of a functional language*
  - *But internals of plug-ins are Fortran or C!*

Segmenting Developer Roles

(diagonalize your matrix)

Developer Roles	Domain Expertise	CS/Coding Expertise	Hardware Expertise
***Application**: Assemble solver modules to solve science problems. (eg. combine hydro+GR+elliptic solver w/MPI driver for Neutron Star simulation)*	Einstein	Elvis	Mort
***Solver**: Write solver modules to implement algorithms. Solvers use driver layer to implement "idiom for parallelism". (e.g. an elliptic solver or hydrodynamics solver)*	Elvis	Einstein	Elvis
***Driver**: Write low-level data allocation/placement, communication and scheduling to implement "idiom for parallelism" for a given "dwarf". (e.g. PUGH)*	Mort	Elvis	Einstein

User/Developer Roles

Developer Roles	Conceptual Model	Instantiation
***Application**: Assemble solver modules to solve science problems.*	Neutron Star Simulation: Hydrodynamics + GR Solver using Adaptive Mesh Refinement (AMR)	BSSN GR Solver + MoL integrator + Valencia Hydro + Carpet AMR Driver + Parameter file (params for NS)
***Solver**: Write solver modules to implement algorithms. Solvers use driver layer to implement "idiom for parallelism".*	Elliptic Solver	PETSC Elliptic Solver pkg. (in C) BAM Elliptic Solver (in C++ & F90) John Town's custom BiCG-Stab implementation (in F77)
***Driver**: Write low-level data allocation/placement, communication and scheduling to implement "idiom for parallelism" for a given "dwarf".*	Parallel boundary exchange idiom for structured grid applications	Carpet AMR Driver SAMRAI AMR Driver GrACE AMR driver PUGH (MPI unigrid driver) SHMUGH (SMP unigrid driver)

Other Contexts for Functional Semantics

- I/O
  - Large shared parallel filesystems are implementing equivalent of cache-coherence protocol at block-level (1 Meg stripes), and that's why it's a pain to scale
  - Relaxed POSIX? (just as impenetrable as relaxed models for cache-coherence)
  - Object DB storage allows you to access and organize data using functional semantics (lambdas, transactions, in-situ operations)
  - Functional semantics offers cleaner approach than Relaxed POSIX to scale storage performance

Problems with Functional Semantics

(if this is so great, then why isn't everyone doing it?)

- **After being reintroduced many times, Functional Languages have not taken root in the mainstream**

- **Performance problems**
  - Locality is enemy of optimal scheduling
  - State recovery complex without quiesced communication (module state can be recovered, but recovering the state that was "in flight" is much more difficult)
  - Overhead of data-rendezvous operations for links on DFG

- **Ergonomic problems (all or nothing)**
  - You have to give up a lot of your favorite gang-of-four design patterns (hash tables a pain)
  - A=B for all time? (hard to grok)
  - Even arrays can be difficult (iStructs)
  - Optimal Scheduling is the enemy of locality
  - State recovery without quiesced communication is difficult (with global memory addressing, even harder)
  - Integration/interoperability with familiar languages + incremental porting path from existing code (directives own this space)
  - *Purity is problematic (as Vijay mentioned)*

Practical Paths to adoption *(lessons from C++)*

- **Coding conventions precede language implementation**
  - example: how did a bastard son of an OO language like C++ win in the open market?
  - programmers used OO design conventions to write C and even assembly code
  - C++ took most commonly used OO design conventions and then turned them in to language constructs
  - But it also didn't force you to adopt *all* of those constructs (you could lapse back to C without penalty)
  - ***Language designers should focus on what functional semantics in use today!***
- **Offer range of implicitly parallel (declarative) semantics**
  - If you adhere to functional semantics, get easier parallelism
  - If you don't adhere to functional semantics, then you pay some penalty in ease of parallelization
- ***Perhaps we are looking for a Dysfunctional Language (C++)***
  - Good for newbies (not all or nothing)
  - Good for huge code bases (incremental porting path)
  - Good for lazy programmers

Hardware Support

- People are already using TBB for concurrent physics
  - Using it despite cost of spinlocks
  - Direct messaging queues would be huge win
- Scratchpads are actually desirable
- Operating Systems / Runtime support
  - Runtimes are mostly dumb
  - That's because OS keeps them in dark (most important functions are privileged)

Conclusions

- If we adopt “asynchronous execution models” as the pancea for all of the ills of SPMD, there is still more medicine to swallow before patient can be cured
- Declarative/Functional Semantics has some great properties for managing asynchronous/stochastic parallelism
- It also creates some problems
  - Scheduling vs. locality management
  - Recovery of state that is in-flight for fault tolerance
  - Load rebalancing for transient imbalances
  - Ergonomically unfriendly research languages
- Take a page from C++ playbook
  - Start with use in practice before elevating to language constructs
  - Provide language that allows incremental path to get functional semantics into broader use

More Material

- Nouveau Dataflow at SIAM PP08
  - Includes Intel Ct, MAGMA/PLASMA, and SISAL
  - http://www1.nersc.gov/projects/SDSA/meetings/SIAM_PP08/
- Dysfunctional Languages
  - http://www.upcrc.illinois.edu/workshops/summit_feb2009/language


NERSC

Is it Time for Change?

- SPMD is easy to reason about (low cognitive load)
  - Bulk-sync communication reduces scope of dataflow hazards that are possible
  - Everybody is *mostly* doing the same thing at the same time

- But all is not well
  - Good SPMD performance creates episodic *flood* communication
  - Requires strict control of noise sources (noise sources are increasing)
  - Exhausting Sources of parallelism through domain decomposition (want to do functional decomposition or concurrent pipelines)

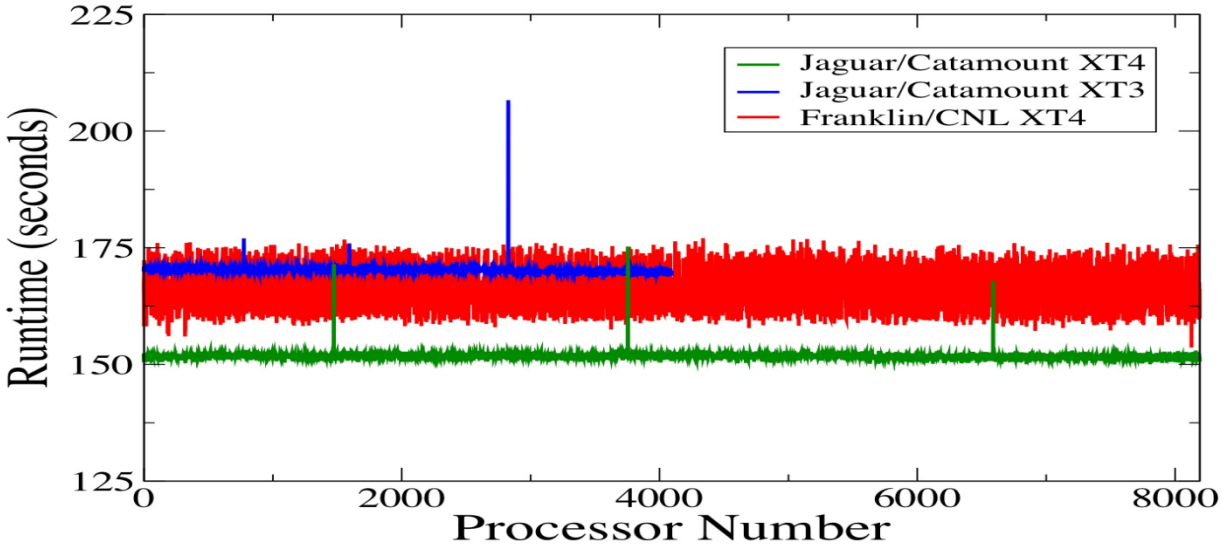
- Async models offer elegant solution
- But if you want to consider asynchronous execution models, then you have a lot more changes to assimilate
  - Current programming models are not easily analyzable (cannot predict static dataflow hazards)

Definitions

- Execution Model: what are the set of hardware mechanisms for making computational progress
  - Asynchronous (aspirational)
  - Dataflow / data-driven
- Programming Model: What are the semantics I use to communicate to underlying exec model
  - Declarative/Constraint-based/Functional model
  - Transactional model
  - SPMD model with global address space (implicit) communication semantics
  - SPMD model with explicit messaging

Increasing Sources of Performance Inhomogeneity Challenge SPMD Model

- Fine grained power management makes even homogeneous cores look heterogeneous
  - *thermal throttling on Sandybridge – no longer guarantee deterministic clock rate*
- Nonuniformities in process technology creates non-uniform operating characteristics for cores on a CMP
- To improve chip yield, disable cores with hard errors
  - *impacts locality of chip-level interconnects*
- Fault resilience introduces inhomogeneity in execution rates
  - *error correction is not instantaneous*
  - *And this will get WAY worse if we move towards software-based resilience*



Post-SPMD Programming Models

- **Requirements: Current models over-specify execution order**
  - Relax scheduling constraints to give flexibility to runtime scheduler for async execution *(more declarative/constraint-based specification of computation)*
  - Non-uniform assignment of work: *feed-forward pipelines that overlap work across heterogeneous processing elements (too hard for humans)*
  - Proactively detect load imbalances: *progress meters instead of global barriers*
  - Locality management: *fixing load imbalance usually hurts locality*

- **Emerging Models that might meet requirements** *(Nouveau Dataflow)*
  - PLASMA and PTP: Dataflow with work-units expressed in Fortran or C
  - CILK: work stealing *(exacerbates locality management problems though)*
  - Ct : SISAL in C++ combined with dynamic dataflow runtime scheduler
  - SWARM: Recently demonstrated for Graph500 benchmark

Conclusion

- **Major outstanding issues**
  - Load imbalance detection without global synchronization
  - Reconciling load imbalance with locality
  - Async scheduling requires strict control of side-effects
  - *Relaxed scheduling constraints will affect bit-reproducibility, which has largely unexplored implications for algorithm stability!*

- **Viable solutions do not look like current practice**
  - Viable solutions are categorically *not* data parallel or SPMD
  - looks more like dataflow *(but there are subtle differences*)

The Questions

1. What are the main benefits of functional programming for parallelism?
2. Should parallelism be implicit?
3. How un-pure can a functional language be before it is not useful?
4. What are the open issues holding up efficient, parallel implementations?
5. How are functional and domain-specific languages related?

Short Answers to the Questions

1. What are the main benefits of functional programming for parallelism?
   - Isolation provides scheduling freedom (hide latency, load imbalance, mapping to novel architectures)
   - Difficult model to teach to fortran and C programmers…
2. Should parallelism be implicit?
   - Explicit (Imperative) approach overspecifies the solution *(limits freedom for compiler and runtime to reorganize the computation and memory layout)*
   - But scheduling freedom can also be a curse (Parry Husband's conundrum)
3. How un-pure can a functional language be before it is not useful?
   - VERY un-pure, and NEEDS to allow some impurity *(impurity makes metals and languages stronger… C++ has broader adoption because it is not rigid OO)*
4. What are the open issues holding up efficient, parallel implementations?
   - Code Generation for machine architectures with unpredictable behavior
   - Functional and declarative parallel languages often make data locality ambiguous
   - Scheduling freedom can create huge scheduling optimization problem
   - **Performance transparency**
5. How are functional and domain-specific languages related?
   - Can make functional constructs palatable to Fortran/C programmers if packaged in domain-specific semantics

What are the benefits of functional semantics for parallelism

- If you know the scope of side effects, many of the hard problems of parallelism become easier
- Safer scheduling (but some things harder)
  - Hide latency
  - Fix load imbalances
  - Express pipeline concurrency or functional partitioning safely
- Easier to determine locality
  - Can facilitate optimizations for locality of reference (automatically time-skew stencils)
  - Automatically transform code to minimize communication cut-set
- Easier to reason about concurrency if dependencies are made explicit
  - Easier to map to non-CC architectures! Mem model

Should Parallelism be Implicit?

- Imperative programming models overdefine implementation
  - Few degrees of freedom available to runtime environment or compiler to improve scheduling of work
  - Mind you this doesn't fix everything with parallel algorithms!
- Declarative (implicit) models for parallelism offer more freedom for implementation
  - But declarative model means we must know scope of side-effects for each unit of computation
  - Functional programming languages make scoping of side-effects well defined
  - However, some commonly used computational Patterns (e.g. gang-of-four) are difficult to express in a "pure" functional language
  - Need to keep a back-door for patterns that don't map well to functional languages *(it's the practical thing to do)*

Scheduling Dangers for Implicit Parallelism

- Can make it difficult to automatically reason about locality unless there is annotation or global analysis
  - NP-hard graph embedding problem
  - **Lack of performacne transparency**
- Scheduling freedom is a benefit and also a curse (Parry Husbands)
  - Unfurl DFG too much and you run out of memory
  - Unfurl too little and you don't expose enough parallelism
  - Unfurl in the wrong direction and you run out of memory and deadlock before exposing enough parallelism
- Implicit parallelism makes scheduling optimization a first order issue
  - leave bread crumbs or hints after previous pass
  - annotate code as to preferred direction of unfurling

How unpure can functional language be?

- C++ shows that a language can be very unpure and be very productive
  - C++ makes CS professors and most computer language architects gag…
  - But it won in the open market
  - Deal with it! (and learn from it)

Lessons of C++

(a little impurity makes metals/languages stronger)

- C++ is broadly adopted because it is unclean
  - Depart from OO if you want to -- not forced to use it everywhere
  - Impurities make languages stronger (and metals too)
  - Godel's law applied to software (broader applicability)
- For implicitly parallel (declarative) languages
  - If you adhere to functional semantics, get easier parallelism
  - If you don't adhere to functional semantics, then you pay some penalty in ease of parallelization
  - Let the programmer decide on a loop or module basis
    - Can't mix semantics in same loop, but can restrict scheduling to sections of code
    - You will expose sufficient parallelism for CMP without global functional guarantee
  - This kind of approach provides an incremental path for migration
  - **WE are looking for a Dysfunctional Language**
    - Good for newbies
    - Good for huge code bases
    - Good for lazy programmers

Observations on Domain-Specific Frameworks and DSL's

- Frameworks and domain-specific languages
  - enforce coding conventions for big software teams
  - Encapsulate a domain-specific "idiom for parallelism"
  - Create familiar semantics for domain experts (more productive)
  - Clear separation of concerns (separate implementation from specification)
- Common design principles for frameworks from SIAM CSE07 and DARPA Ogden frameworks meeting
  - Give up main(): *schedule controlled by framework*
  - Stateless: *Plug-ins only operate on state passed-in when invoked*
  - Bounded (or well-understood) side-effects: *Plug-ins promise to restrict memory touched to that passed to it (same as CILK)*
- *By gosh, these are attributes of a functional language*
  - *But internals of plug-ins are Fortran or C!*

Conundrum

- Frameworks are very limited in scope
  - A good framework targets a specific space of problems (not everything in the world)
  - Not general enough: not scalable to deploy frameworks as a solution
  - Difficult to optimize parallel constructs along with code
- Languages can be deployed in a scalable manner
  - Unclear which basic constructs of parallel language are broadly applicable
  - Need practical experience to filter good from bad constructs
  - Usually gain this experience using framework to implement “idiom for parallelism” (in lieu of pre-existing language construct)
- We have a chicken-and-egg problem here
  - How do we determine what constructs to put in a language?

More Lessons of C++

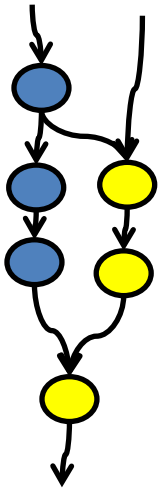
(should coding conventions precede language constructs?)

- OO software practices for C did not begin with C++
  - Programmers were using OO design conventions to write C applications (and even assembly code)
  - C++ took most commonly used OO design conventions and turned them into language constructs

- Should design conventions for parallelism precede language implementation?
  - Offer constructs first as design patterns (suboptimal performance)
  - Winning patterns get hoisted into language constructs

Why Functional Languages are Good/Bad

- Controlling side-effects makes scheduling decisions easier
  - Easier to map onto complex hardware
  - Easier to hide latency or rearrange operations to avoid latency-critical communication regions (slack)
- Functional requirement that A=B for all time is difficult to grok
  - Programmers don't think that way (think of program phases of execution)
  - Need to be able to localize side-effect constraints to individual loops, solvers, or program modules
    - Per-loop basis with stream programming
    - Strong-typing to control variable access (changeable over time)
    - Ct TVEC: nice middle-of-the-road guarantees for Single-assigment array (much better than istructs)

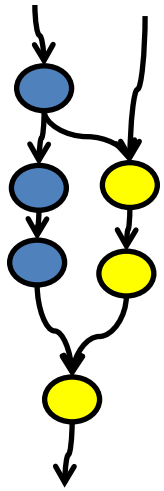


Whats wrong with functional Languages

- Purity is the biggest impediment to adoption
  - A=B for all time in pure functional language!
  - Programmers don't think that way
  - They think in modules, loop, or "solver" boundaries
  - Global scope of func lang side-effect guarantees is too broad
- Relaxing constraints
  - Restrict scope of side-effect guarantees to loops or modules of program (user-defined scope)
    - Restricts scheduler freedom, but makes reasoning easier
  - Only need to restrict (or define) scope of side-effects to get benefits in many cases
    - don't need to completely eliminate side-effects to get sched benefits
  - Allow programmer to decide when to adhere to functional model and when not to (same as C++ for object orientation)
    - not mixed within same module
    - Programmer loses parallelism benefits, but can pick and choose when to trade against ease of expression

Implicit Parallelism

- Requires bounding of side-effects so that you know what is a legal schedule of operations
- Functional approach offers precise restriction of side-effects that are true for “entire execution” (A=B)
  - May be a bit too rigid for some code examples
  - Sometimes only need to bound side-effects (not eliminate)
- Programmers think in terms of program phases and modules
  - Global guarantees are hard to fathom
  - Istructures are an abomination (Ct TVEC is better)
- Example of weaker guarantee:
  - Stream programming: Scope side-effects-free guarantee to loop boundaries
  - Ct TVECs (better than iStructures)



Other Lessons from C++

- C++ is very “un-pure” compared to other OO languages
  - This is as much an asset as a curse
  - Enables incremental porting (easy to mix OO with non-OO code)
  - Does not prevent programmer from shooting self in foot or writing indecipherable code
- Functional language needs to be flexible
  - Need to allow un-pure implementation (mixing of pure and unpure)
    - For incremental porting
    - Also, functional style can be too rigid for some constructs (allow programmer to decide where to get benefit of isolation or not)
    - Cannot mix constructs within a loop or module
    - Streaming is example of guaranteeing isolation only within individual loops
  - Programmers don’t want A=B to be true for entire program (just in unit of program they are immediately concerned with)
    - For parallelism we don’t need to prove isolation (lack of side-effects) across entire program for all time
    - Understand scope of side-effects within loop or program module rather than all time

From Object Oriented Programming to C++

- We were using object oriented “style” of programming in C long before emergence of C++
  - C++ did not result in object-oriented coding practice, rather standard coding practices become encoded in C++
  - C++ took years of practical experience with software engineering and encapsulated the most broadly-used constructs
  - Experience first in software frameworks was migrated into language
- How can we do this for parallel constructs?
  - Start with point-experiments encoded in frameworks
  - Take “winning” (broadly applicable) constructs and migrate into language?

Frameworks

Managing code complexity

Facilitating multi-physics coupling

Not a replacement for languages

Application Code Complexity

- **Application Complexity has Grown**
  - Big Science on leading-edge HPC systems is a multi-disciplinary, multi-institutional, multi-national efforts! *(and we are not just talking about particle accelerators and Tokamaks)*
  - Looking more like science on atom-smashers

- **Advanced Parallel Languages are Necessary, but NOT Sufficient!**
  - Need higher-level organizing constructs for teams of programmers
  - Languages must work together with frameworks for a complete solution!

Role of Community Codes & Frameworks

- Clearly separate roles and responsibilities of your expert programmers from that of the domain experts/scientist/users (productivity layer vs. performance layer)
- Define a *social* contract between the expert programmers and the domain scientists
- Enforces and facilitates SW engineering style/discipline to ensure correctness
- Hides complex domain-specific parallel abstractions from scientist/users to enable performance (hence, most effective when applied to community codes)
- Allow scientists/users to code nominally serial plug-ins that are invoked by a parallel “driver” (either as DAG or constraint-based scheduler) to enable productivity

Segmenting Developer Roles

(diagonalize your matrix)

Developer Roles	Domain Expertise	CS/Coding Expertise	Hardware Expertise
***Application**: Assemble solver modules to solve science problems. (eg. combine hydro+GR+elliptic solver w/MPI driver for Neutron Star simulation)*	Einstein	Elvis	Mort
***Solver**: Write solver modules to implement algorithms. Solvers use driver layer to implement "idiom for parallelism". (e.g. an elliptic solver or hydrodynamics solver)*	Elvis	Einstein	Elvis
***Driver**: Write low-level data allocation/placement, communication and scheduling to implement "idiom for parallelism" for a given "dwarf". (e.g. PUGH)*	Mort	Elvis	Einstein

Framework User/Developer Roles

Developer Roles	Conceptual Model	Instantiation
***Application**: Assemble solver modules to solve science problems.*	Neutron Star Simulation: Hydrodynamics + GR Solver using Adaptive Mesh Refinement (AMR)	BSSN GR Solver + MoL integrator + Valencia Hydro + Carpet AMR Driver + Parameter file (params for NS)
***Solver**: Write solver modules to implement algorithms. Solvers use driver layer to implement "idiom for parallelism".*	Elliptic Solver	PETSC Elliptic Solver pkg. (in C) BAM Elliptic Solver (in C++ & F90) John Town's custom BiCG-Stab implementation (in F77)
***Driver**: Write low-level data allocation/placement, communication and scheduling to implement "idiom for parallelism" for a given "dwarf".*	Parallel boundary exchange idiom for structured grid applications	Carpet AMR Driver SAMRAI AMR Driver GrACE AMR driver PUGH (MPI unigrid driver) SHMUGH (SMP unigrid driver)

Observations on Domain-Specific Frameworks and Embedded DSL's

- Frameworks and domain-specific languages
  - enforce coding conventions for big software teams
  - Encapsulate a domain-specific "idiom for parallelism"
  - Create familiar semantics for domain experts (more productive)
  - *Clear separation of concerns (separate implementation from specification)*
- Common design principles for frameworks from SIAM CSE07 and DARPA Ogden frameworks meeting
  - Give up main(): *schedule controlled by framework*
  - Stateless: *Plug-ins only operate on state passed-in when invoked*
  - Bounded (or well-understood) side-effects: *Plug-ins promise to restrict memory touched to that passed to it (same as CILK)*

What are *some* of the Problems?

- **We are addicted to SPMD execution model**
  - Homogeneous code == low cognitive load *(good for human beings)*
  - But… Heterogeneous cores will execute heterogeneous code
  - Strong scaling also motivates feed-forward pipelines *(heterogeneous exec)*
  - Bulk synchronous model is out of question if noise sources increase

- **Work assignment / load balancing**
  - How to automatically adjust to different balance of components in current and future architectures (rewrite?)
  - Identify load imbalance in time to react (latency of state migration)
  - Reconcile load imbalance and work-assignment with locality

- ***These are classic unsolved problems of CS (be afraid!)***

Trouble on the Horizon

- **New sources of inhomogeneity in hardware**
  - Sparing redundant resources to tolerate hard errors creates inhomogeneous communication characteristics
  - constrained interconnect topologies (graph embedding for comm topology)
  - Inhomogeneity in process technology leads to non-uniform clock rates
  - Thermal Throttling (Intel Sandybridge)
  - Hardware fault recovery to tolerate transient errors (software recovery mechanisms will make it even worse)
- **Algorithm/Application requirements**
  - Adaptive Algorithms/AMR: Fastest, most energy efficient FLOP is the one you don't execute
  - Irregular structure: Irregular mesh, Sparse matrix, and MD computations have irregular work and dependency patterns (DAG scheduling & Curt)
  - Irregular work: Subcycling for ODEs for combustion chemistry or to squeeze out residual error for fluids probs creates inhomogeneity for regular structures
  - Exhausted parallelism through domain decomposition motivates move towards functional decomposition (climate coupler & Mike Heroux)
  - Software Engineering: Separation of concerns for frameworks & libraries
- **New Memory Hierarchies and structures**
  - Non-coherent Global Address Space or CC with relaxed consistency: when do I sync?
  - Disjoint memories: marshalling/unmarshalling data for accelerators & scratchpads