Some Do's and Don'ts for Designing Parallel Languages

Laxmikant (Sanjay) Kale

http://charm.cs.illinois.edu

Our Focus Area: CSE in HPC

- BigData… lets keep it on the side for this discussion..
  - It turns out many ideas in CSE/hpc will help the broadened big-data scenarios as well
- CSE apps are characterized by:
  - Iterative computations
  - Persistence in behavior
    - Even for dynamically adaptive applications
  - A relatively small repertoire of data structures
    - Structured/unstructured meshes, matrices, particles, hierarchical trees, ..

State of the field: Applications

- Strong scaling needs
  - Since early days, until recently, if you get a larger machine, you increased the resolution
  - Now, increasingly: we need to solve the same (resolution) problem, but faster
- Multi-physics applications
- Multi-module applications
- Multi-scale applications

State of the field: Architecture

- Frequency increases stopped in 2003
  - Stabilized around 3GHz
  - Reason: thermal
  - Power dissipation of a chip can’t be much more than 100 W
- Moore’s law continues:
  - 22nm exists, 14nm around the corner
  - Limits is somewhere around 5 nm
  - That’s only 50A
- Consequence:
  - 30-50B transistors per processor chip
  - Many powerful cores
  - Or many many somewhat less powerful cores

Exascale Challenges

- Main challenge: variability
  - Static/dynamic
  - Heterogeneity: processor types, process variation, ..
  - Power/Temperature/Energy
  - Component failure

New Languages: acceptance?

- The old attitude: disdain
  - For good reason
  - The next 700 languages
- History:
  - Fortran 1955
  - Algol/Pascal: 1960s
  - C: 1970s
  - C++: 1980s
  - Java: 1990s
  - Interpretive/scripting languages: Python, TCl/Tk, Ruby…
  - Newer crop: Go, ..
- So, its hard to get a new language accepted

The newfound acceptance of languages in HPC

- The challenges headlined by exascale
- Examples:
  - X10
  - Chapel
  - Legion
  - All the new task models: Parsec, OmpSS, Openmp task model
- Within US DoE:
  - serious evaluation of new programming models

Outline of the talk

- So, you want to design a new language
- Here is some advice from a old hand
- I will outline a few do’s and don’ts
- To begin with: some design principles:

Aim NOT for full automation,
But for a good division of labor
between the programmer and the system

Example of Full automation

- Parallelizing compiler?
  - Full automation? Not really: only if you start from a sequential program
  - But still, why not?
  - After 45 years of research
    - Some very good intellectual successes
    - But not enough

Avoid Pie in the Sky approaches,
Bottom up development of abstractions

Corollary: Adaptive Runtime

- Build on top an Adaptive Runtime System
- Programmers can decide what to do in parallel relatively easily
- But resource management?
  - i.e. which processor does what and when,
  - Which processor has which data
  - Is tedious and automatable
- Today I see no reason to decide develop a higher level language without using a RTS
  - And frankly, nothing better than my group's Charm++ ☺

Adaptive Runtime Systems

- What is an Adaptive Runtime System?
  - It *observes* what is going on in a parallel computation on a given machine
    - feedback from the machine and the application
  - And then
  - Takes actions to *control* the system, so its executing more efficiently
- How to empower an Adaptive Runtime System?

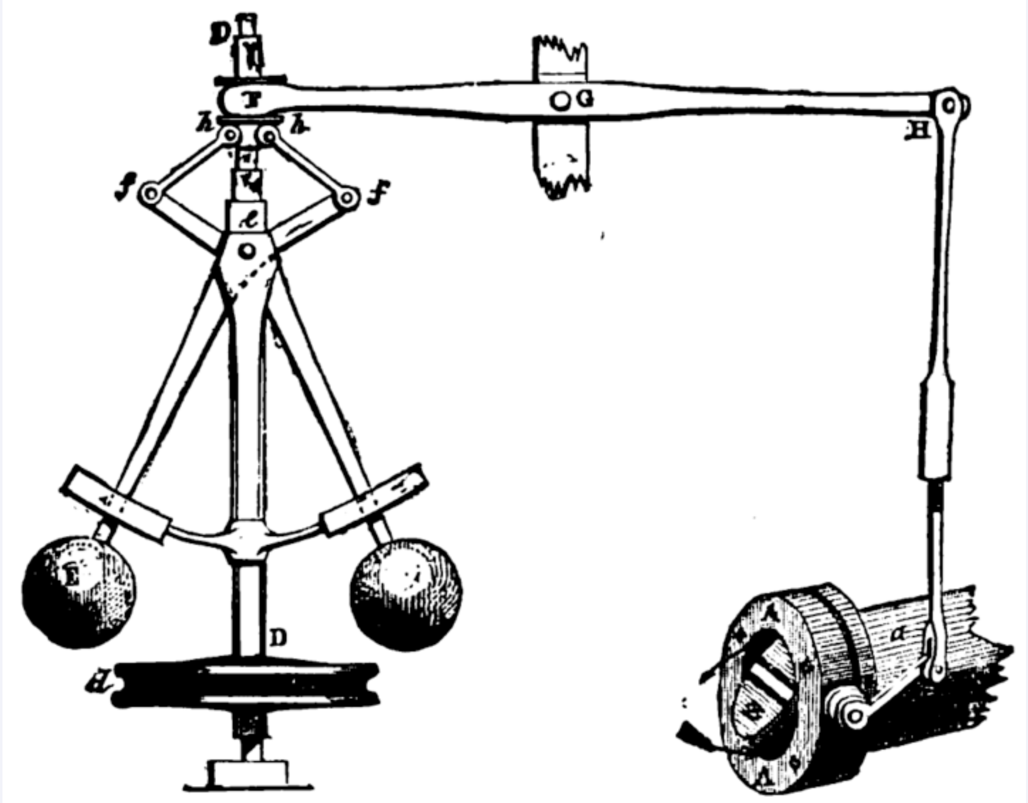
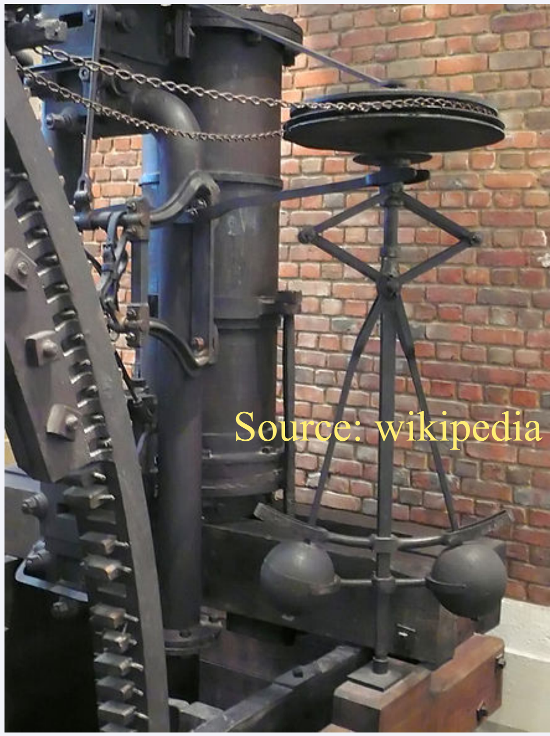


FIG. 4.—*Governor and Throttle-Valve.*

Source: Wikipedia

Governors

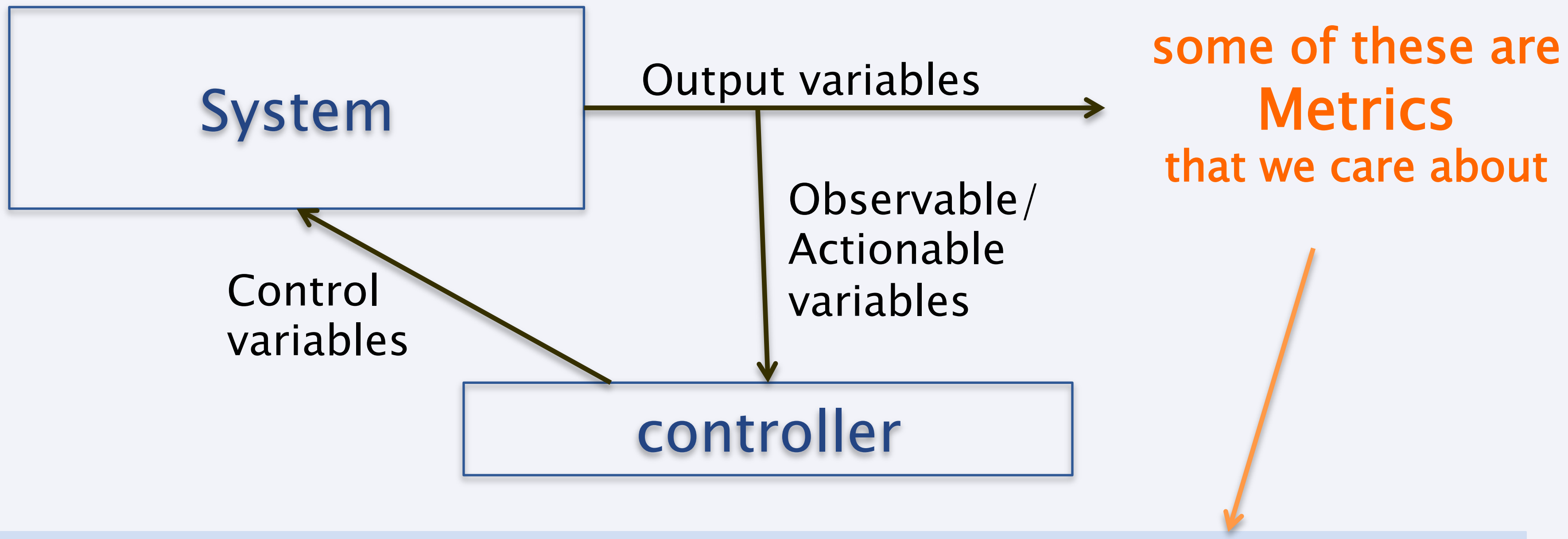
- Around 1788 AD, James Watt and Mathew Boulton solved a problem with their steam engine
  - They added a cruise control… well, RPM control
  - How to make the motor spin at the same constant speed
  - If it spins faster, the large masses move outwards
  - This moves a throttle valve so less steam is allowed in to push the prime mover



Control theory, Maxwell, ..

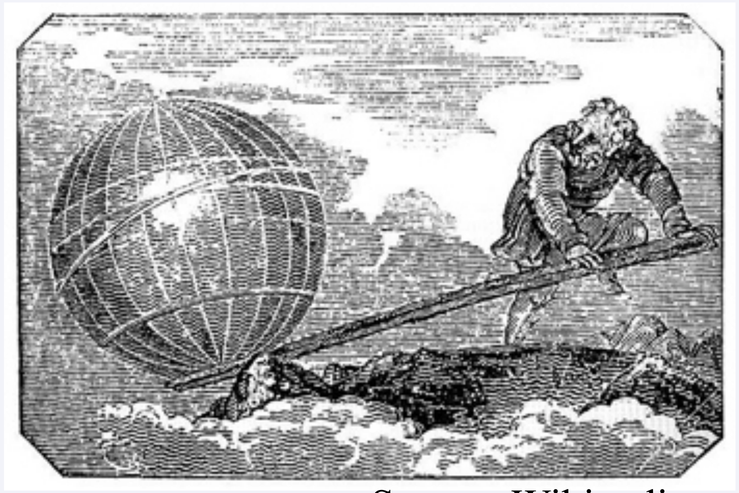
- You let the system "misbehave", and use that misbehavior to correct it..
- The control theory was concerned with stability, and related issues
  - Fixed delay makes for highly analyzable system with good math demonstration
- We will just take two related notions:
  - Controllability
  - Observability
- And stretch them a bit for our purposes

A modified system diagram



These include one or more:

- Objective functions (minimize, maximize, optimize)
- Constraints: “must be less than”, ..


Source: Wikipedia

Archimedes is supposed to have said, of the lever:
Give me a place to stand on,
and I will move the Earth

Where do you get controllable and observables in parallel computations?

My Mantra for empowering RTS

O M

My Mantra

a

$\mathcal{OM}$

My Mantra

Oh….Maybe the order doesn't matter

O M a

My Mantra

OaM

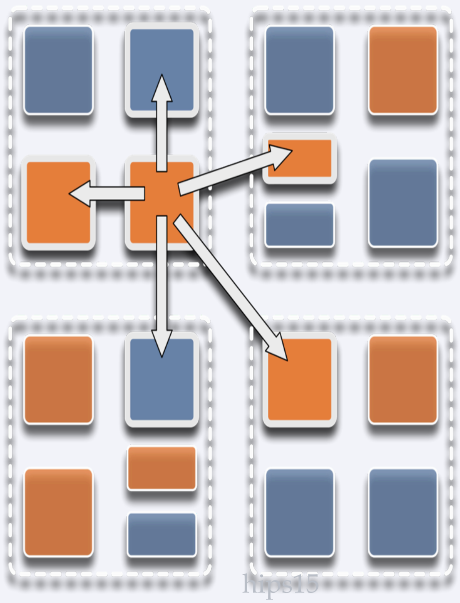
verdecomposition

synchrony

igratability

Overdecomposition

- Decompose the work units & data units into many more pieces than execution units
  - Cores/Nodes/..
- Not so hard: we do decomposition anyway



Migratability

- Allow these work and data units to be migratable at runtime
  - i.e. the programmer or runtime, can move them
- Consequences for the app-developer
  - Communication must now be addressed to logical units with global names, not to physical processors
  - But this is a good thing
- Consequences for RTS
  - Must keep track of where each unit is
  - Naming and location management

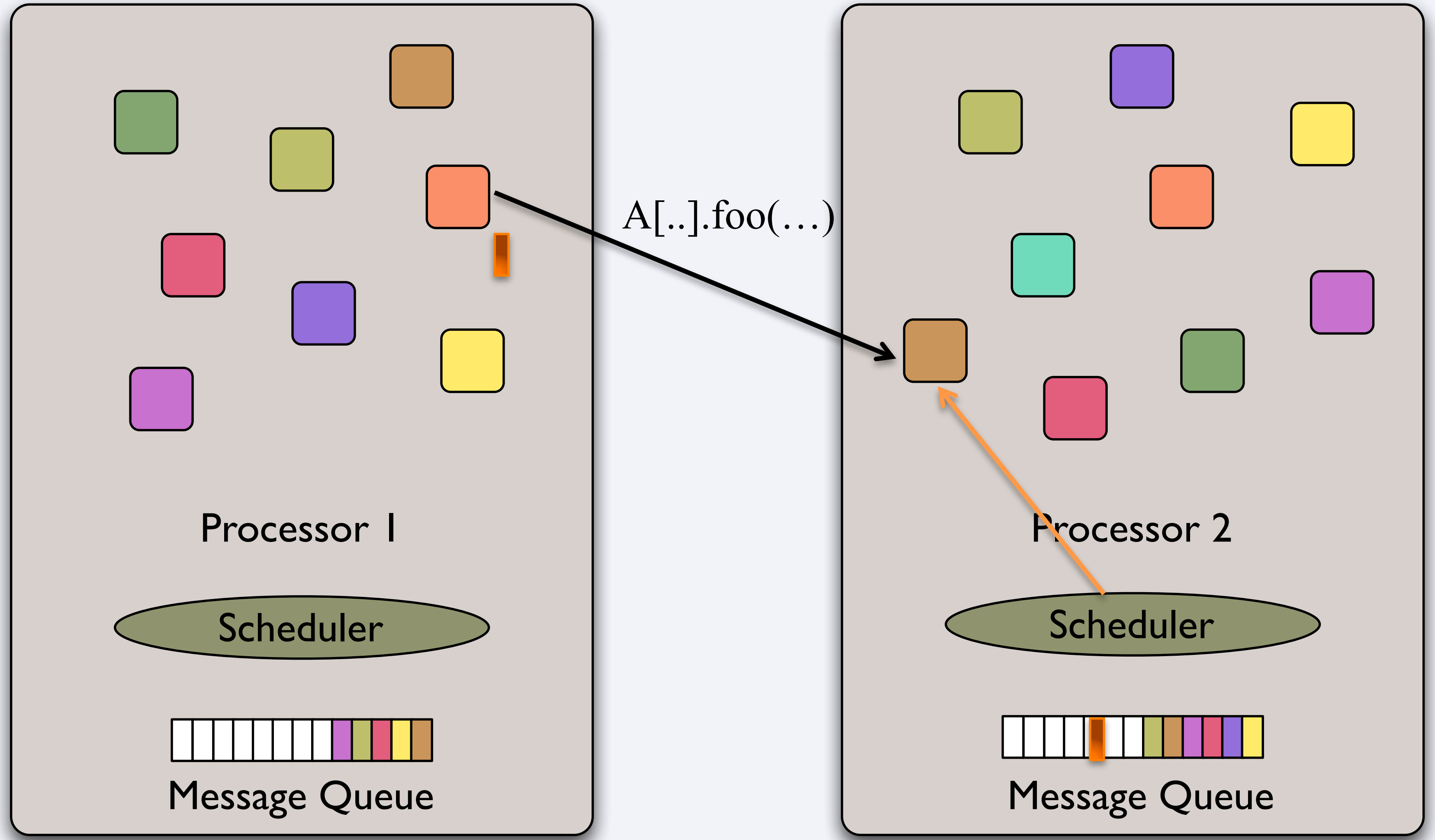
Asynchrony: Message-Driven Execution

- Now:
  - You have multiple units on each processor
  - They address each other via logical names
- Need for scheduling:
  - What sequence should the work units execute in?
  - One answer: let the programmer sequence them
    - Seen in current codes, e.g. some AMR frameworks
  - Message-driven execution:
    - Let the work-unit that happens to have data ("message") available for it execute next
    - Let the RTS select among ready work units
    - Programmer should not specify what executes next, but can influence it via priorities

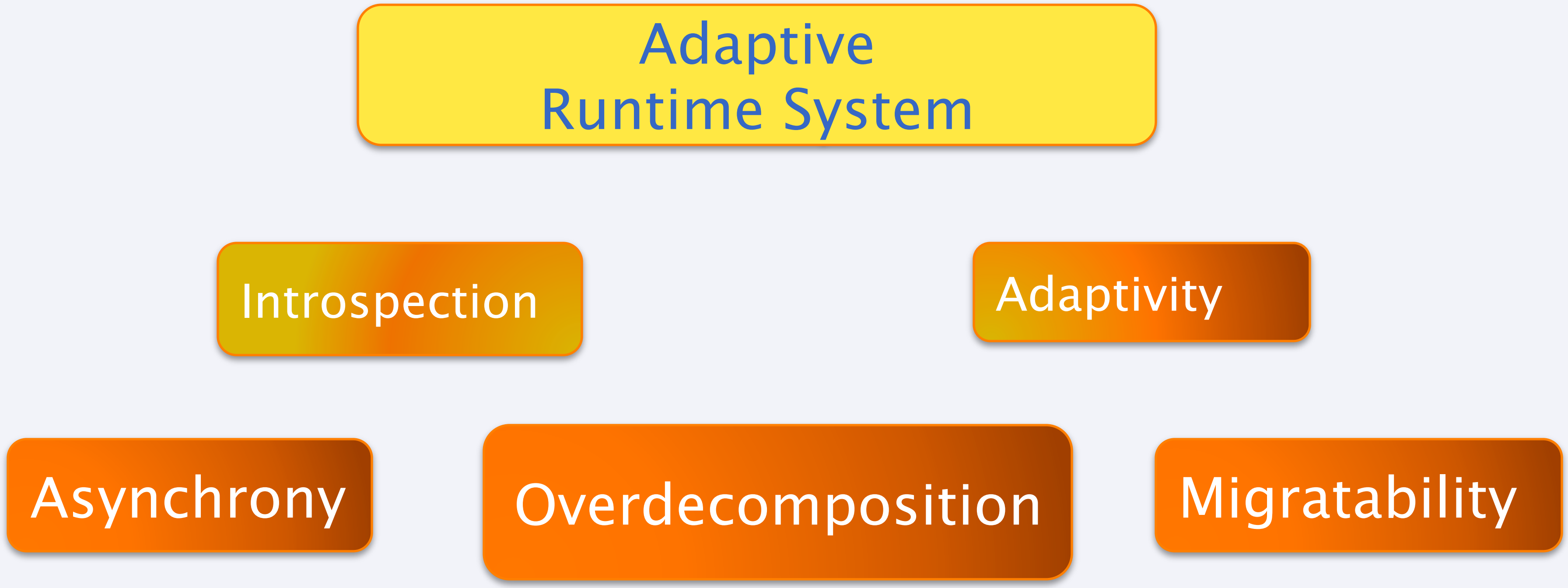
Charm++

- Objects, called chares:
  - Organized into multiple collections, each with its own indexing
  - Asynchronous method invocations
- User-level "run" threads embedded in chares
- Asynchronous (non-blocking) reductions
- "structured dagger":
  - script-like notation to express dependencies among computations and messages within chares

Message-driven Execution



Empowering the RTS



- The Adaptive RTS can:
  - Dynamically balance loads
  - Optimize communication:
    - Spread over time, async collectives
  - Automatic latency tolerance
  - Prefetch data with almost perfect predictability

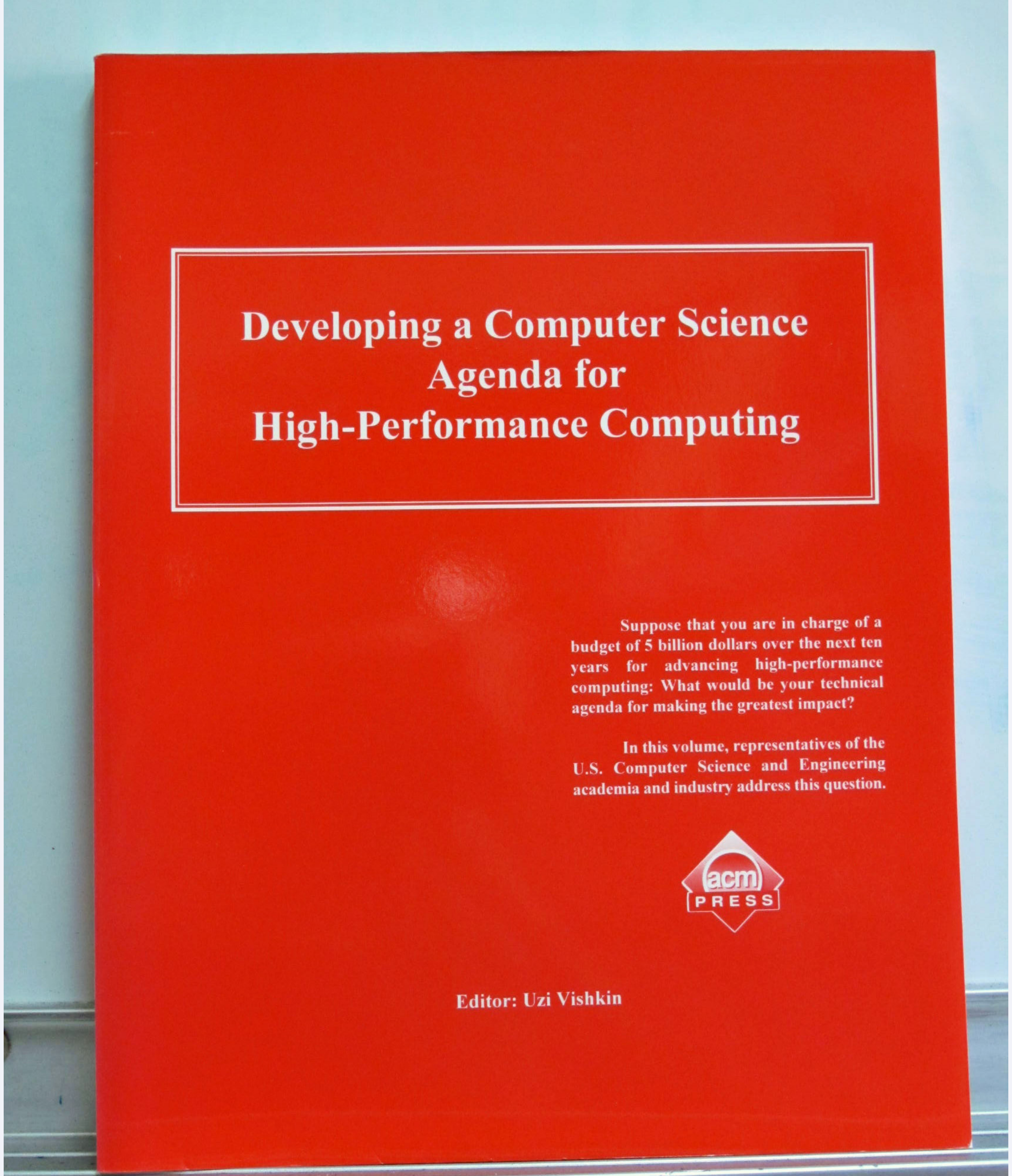
So, specific prescription

- Build you HLL on top of an adaptive runtime system based on overdecomposition, asynchrony and migratability
- Currently, that is just Charm++
- New systems are being designed
  - OCR, etc.
  - But will be very similar, in my opinion, to Charm++ RTS
  - (not necessarily Charm++ "language")

Develop parallel Languages via Application -Oriented but Computer Science centered research

Computer Scientists' role in HPC

- We computer scientists tend to be "platonic"
  - Pursue an idea just because its "beautiful"
  - Ignoring needs of practical science/engineering applications
- Alternatively:
  - Worked on a single application … essentially as programmers!
  - But that doesn't lead to broad enabling technology
- What is needed:
  - Application *oriented,* yet computer science *centered* research
  - Work on multiple applications,
  - Develop abstractions triggered by needs of one, but in a way that's useful for many
  - Accrete abstractions in practical parallel software systems



Developing a Computer Science Agenda for High-Performance Computing

Suppose that you are in charge of a budget of 5 billion dollars over the next ten years for advancing high-performance computing: What would be your technical agenda for making the greatest impact?

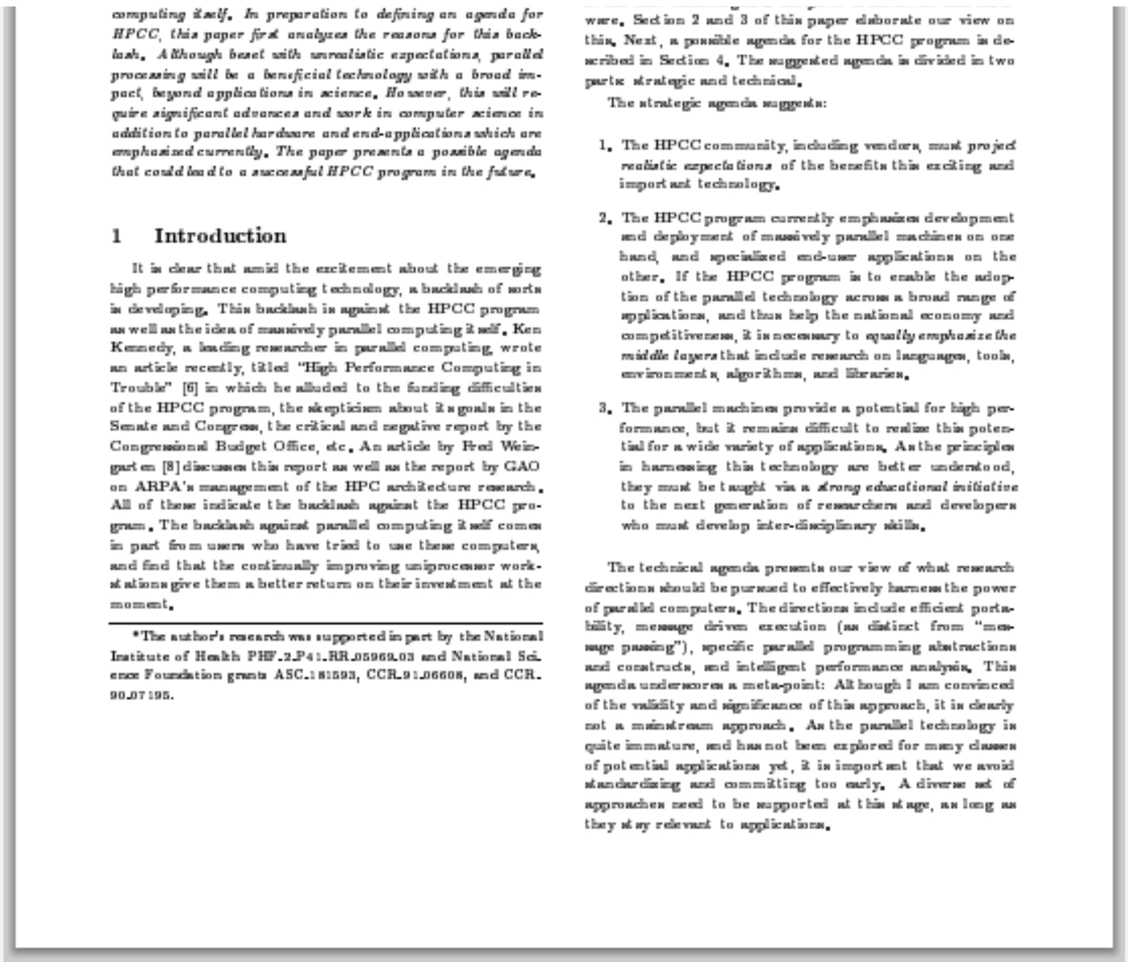
In this volume, representatives of the U.S. Computer Science and Engineering academia and industry address this question.

acm PRESS

Editor: Uzi Vishkin

APPLICATION ORIENTED AND COMPUTER SCIENCE CENTERED HPCC RESEARCH

Laxmikant V. Kalé
Department of Computer Science
University of Illinois
Urbana, IL 61801
E-mail: kale@cs.uiuc.edu



computing itself. In preparation to defining an agenda for HPCC, this paper first analyzes the reasons for this backlash. Although beset with unrealistic expectations, parallel processing will be a beneficial technology with a broad impact, beyond applications in science. However, this will require significant advances and work in computer science in addition to parallel hardware and end-applications which are emphasized currently. The paper presents a possible agenda that could lead to a successful HPCC program in the future.

1 Introduction

It is clear that amid the excitement about the emerging high performance computing technology, a backlash of sorts is developing. This backlash is against the HPCC program as well as the idea of massively parallel computing itself. Ken Kennedy, a leading researcher in parallel computing, wrote an article recently, titled "High Performance Computing in Trouble" [6] in which he alluded to the funding difficulties of the HPCC program, the skepticism about its goals in the Senate and Congress, the critical and negative report by the Congressional Budget Office, etc. An article by Fred Weingarten [8] discusses this report as well as the report by GAO on ARPA's management of the HPC architecture research. All of these indicate the backlash against the HPCC program. The backlash against parallel computing itself comes in part from users who have tried to use these computers, and find that the continually improving uniprocessor workstations give them a better return on their investment at the moment.

*The author's research was supported in part by the National Institute of Health PHF-2-P41-RR-05969-03 and National Science Foundation grants ASC-181593, CCR-91-06608, and CCR-90-07195.

were. Section 2 and 3 of this paper elaborate our view on this. Next, a possible agenda for the HPCC program is described in Section 4. The suggested agenda is divided in two parts: strategic and technical.

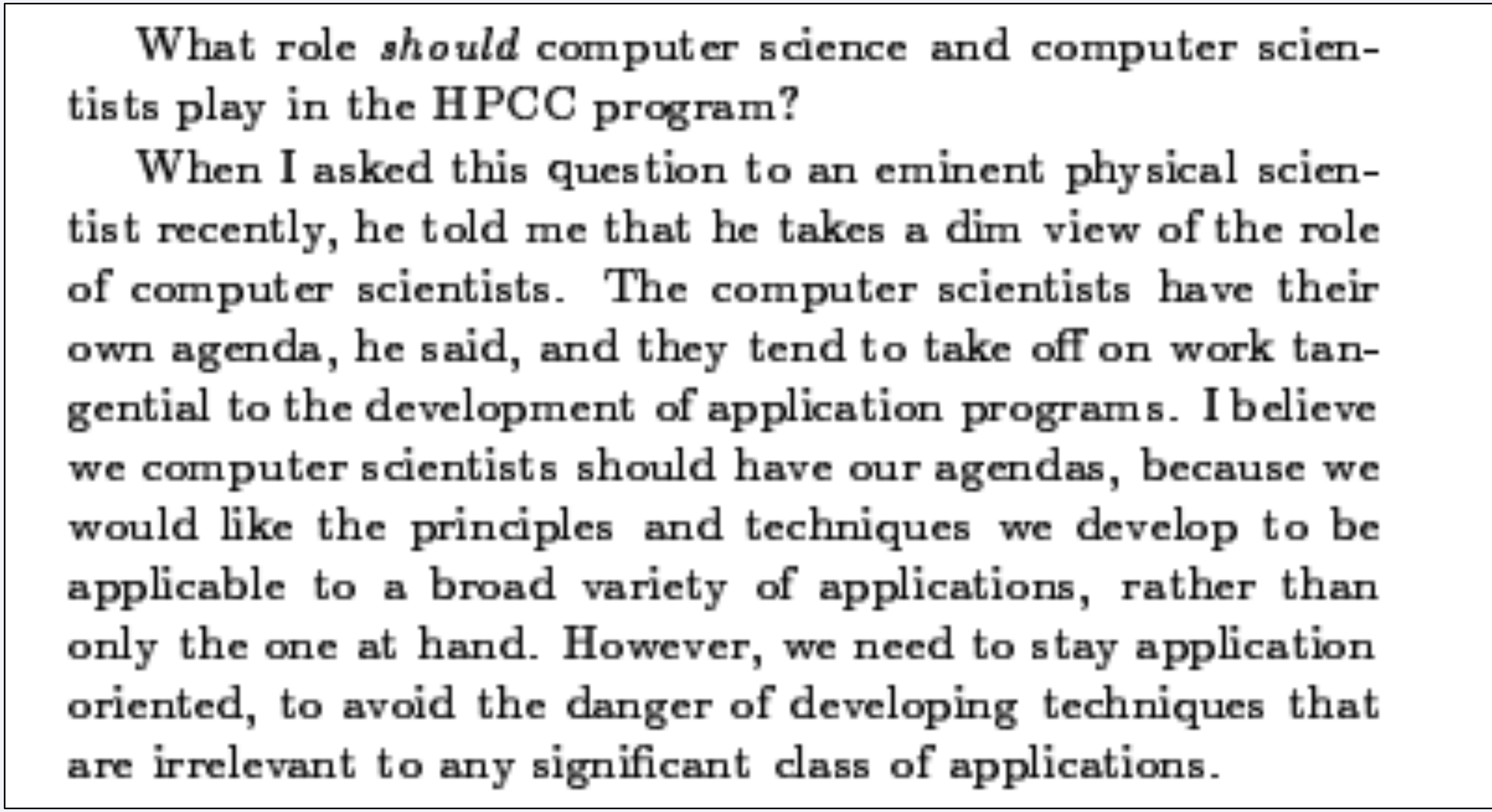
The strategic agenda suggests:

1. The HPCC community, including vendors, must *project realistic expectations* of the benefits this exciting and important technology.

2. The HPCC program currently emphasizes development and deployment of massively parallel machines on one hand, and specialized end-user applications on the other. If the HPCC program is to enable the adoption of the parallel technology across a broad range of applications, and thus help the national economy and competitiveness, it is necessary to *equally emphasize the middle layers* that include research on languages, tools, environments, algorithms, and libraries.

3. The parallel machines provide a potential for high performance, but it remains difficult to realize this potential for a wide variety of applications. As the principles in harnessing this technology are better understood, they must be taught via a *strong educational initiative* to the next generation of researchers and developers who must develop inter-disciplinary skills.

The technical agenda presents our view of what research directions should be pursued to effectively harness the power of parallel computers. The directions include efficient portability, message driven execution (as distinct from "message passing"), specific parallel programming abstractions and constructs, and intelligent performance analysis. This agenda underscores a meta-point: Although I am convinced of the validity and significance of this approach, it is clearly not a mainstream approach. As the parallel technology is quite immature, and has not been explored for many classes of potential applications yet, it is important that we avoid standardizing and committing too early. A diverse set of approaches need to be supported at this stage, as long as they stay relevant to applications.



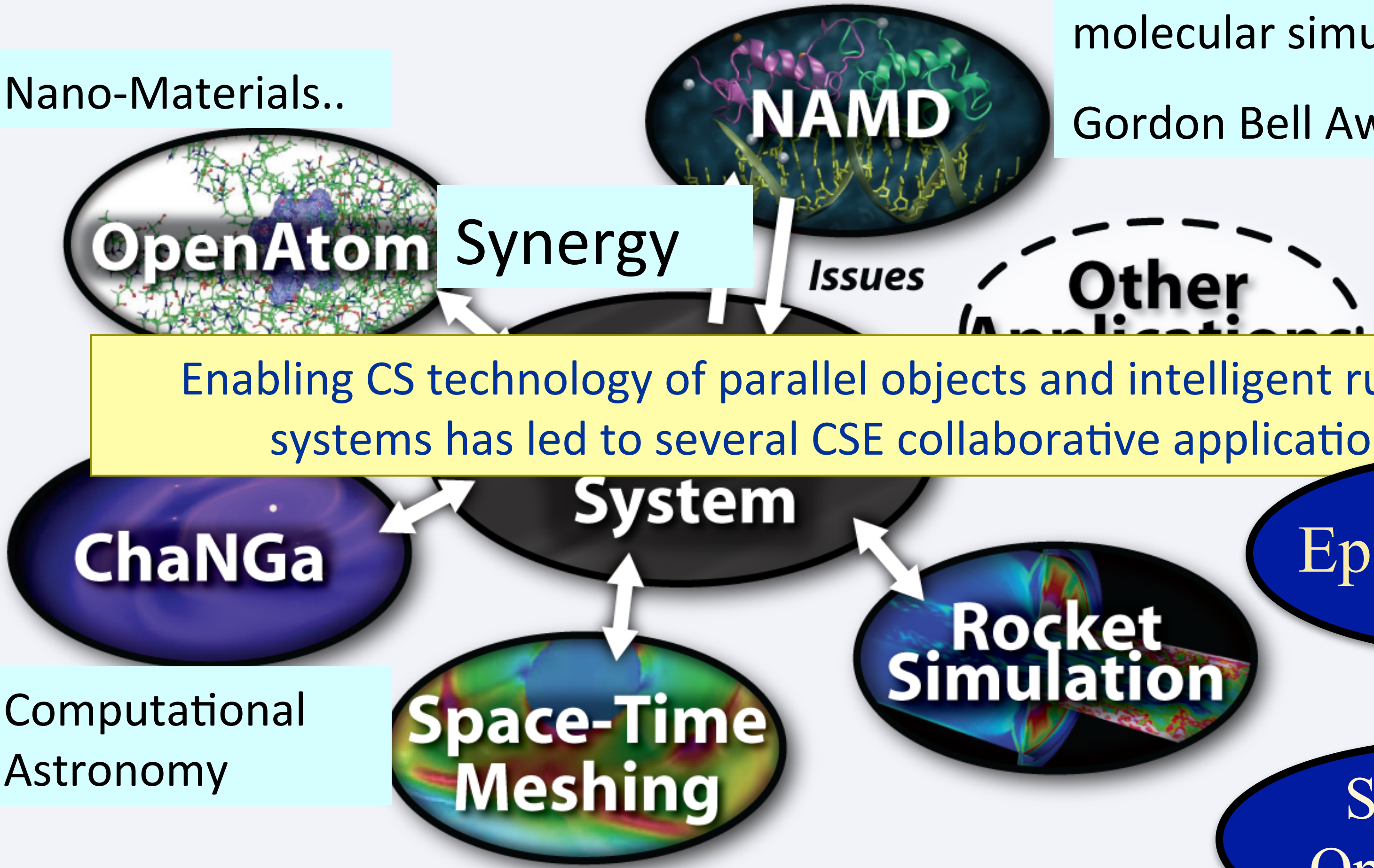
What role *should* computer science and computer scientists play in the HPCC program?

When I asked this question to an eminent physical scientist recently, he told me that he takes a dim view of the role of computer scientists. The computer scientists have their own agenda, he said, and they tend to take off on work tangential to the development of application programs. I believe we computer scientists should have our agendas, because we would like the principles and techniques we develop to be applicable to a broad variety of applications, rather than only the one at hand. However, we need to stay application oriented, to avoid the danger of developing techniques that are irrelevant to any significant class of applications.

So, Prescription:

- Design abstractions based solidly on use-cases
  - Application-oriented yet computer-science centered approach
- Motivate language design by multiple application use-cases
- Test and hone them in the context of multiple full-fledged applications
- Anecdote about an HLL designer

Charm++ and CSE Applications



Well-known Biophysics molecular simulations App

Gordon Bell Award, 2002

Nano-Materials..

Synergy

Enabling CS technology of parallel objects and intelligent runtime systems has led to several CSE collaborative applications

EpiSimdemics

Computational Astronomy

Stochastic Optimization

Next, Syntax

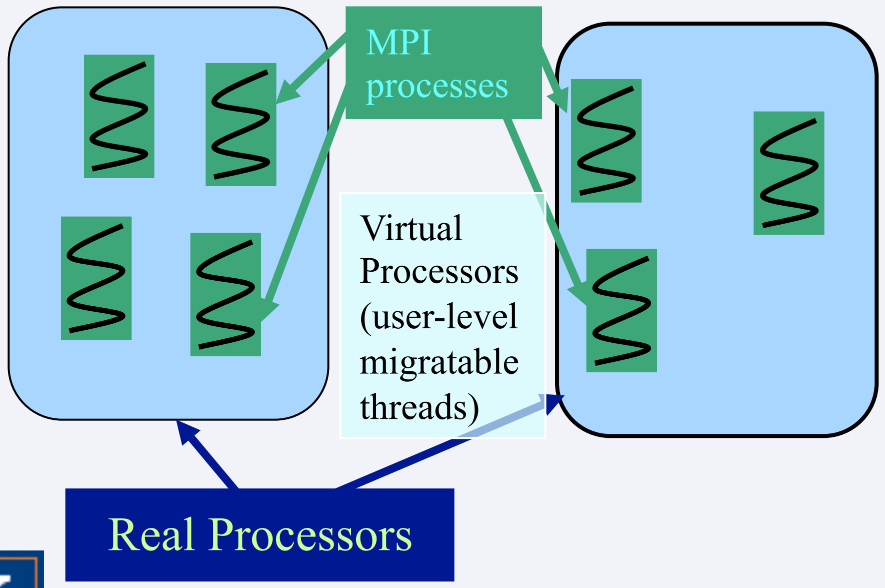
- Is syntax (and syntactic sugar) important?
- Yes, but..
- Alan Perlis: *Too much syntactic sugar gives you cancer of the semicolon*
  - (This from a Lisp proponent! Proliferator of parenthesis)
- Syntax prescriptions:
  - No gratuitous syntax invention
  - For well-established concepts, stick to norms
  - Add it where it provides true convenience, avoids boilerplate, or clarifies meaning

Compiler Support

- Compiler supported language vs a library-like "language"
- Tradeoff:
  - Compilation and static analysis facilitates a lot more optimization, and boilerplate ellimination
  - But you have to buy into a flexible compiler infrastructure
  - (as an aside: you want to stay away form taking responsibility for back-end optimization code generation)

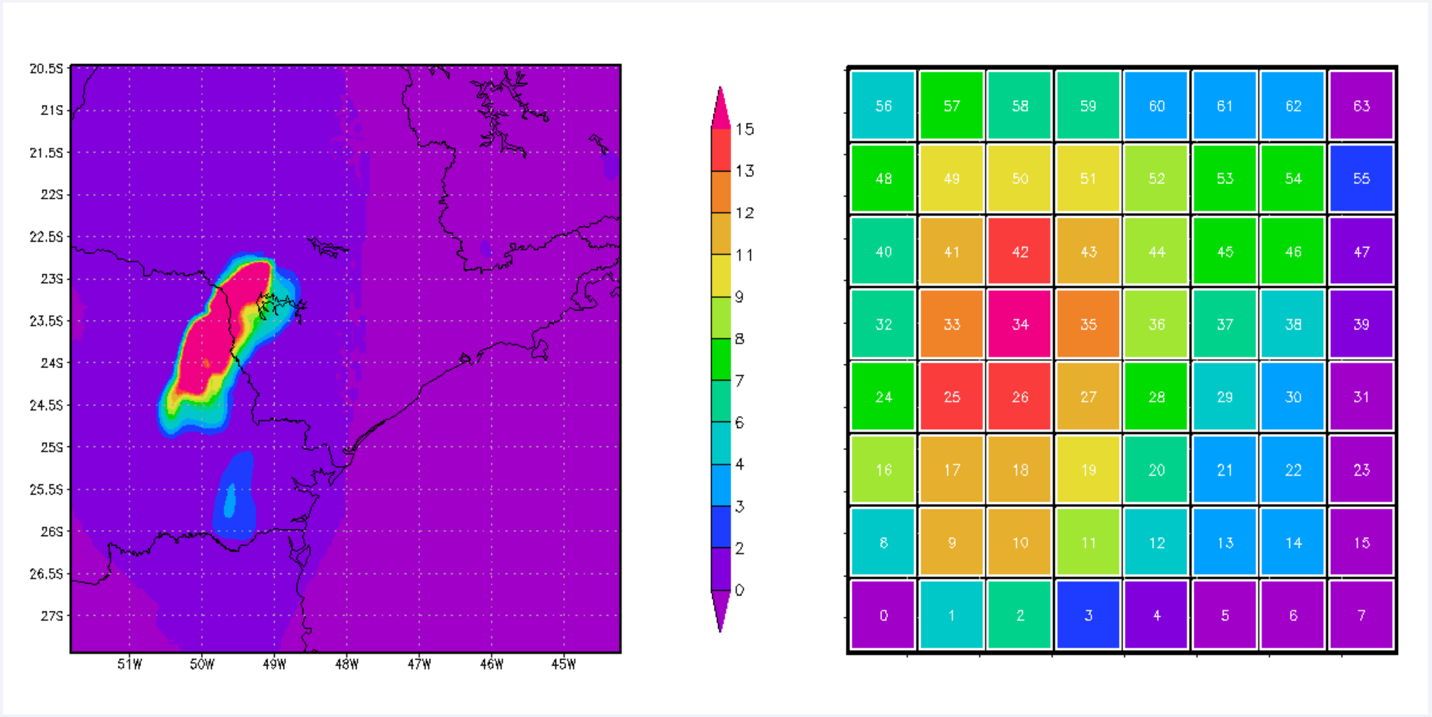
AMPI: Adaptive MPI

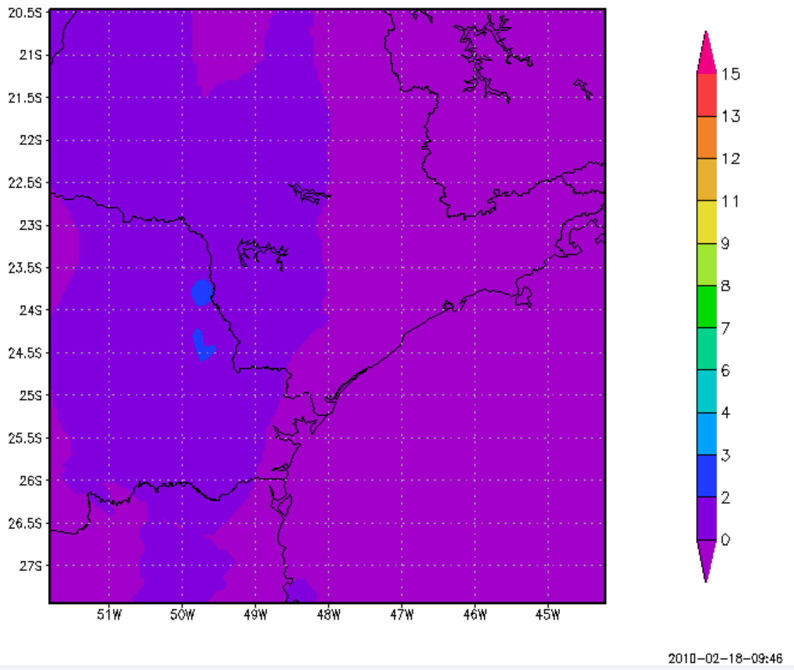
- Each MPI process is implemented as a user-level thread
- Threads are light-weight and migratable!
  - <1 microsecond context switch time, potentially >100k threads per core
- Each thread is embedded in a Charm++ object (chare)

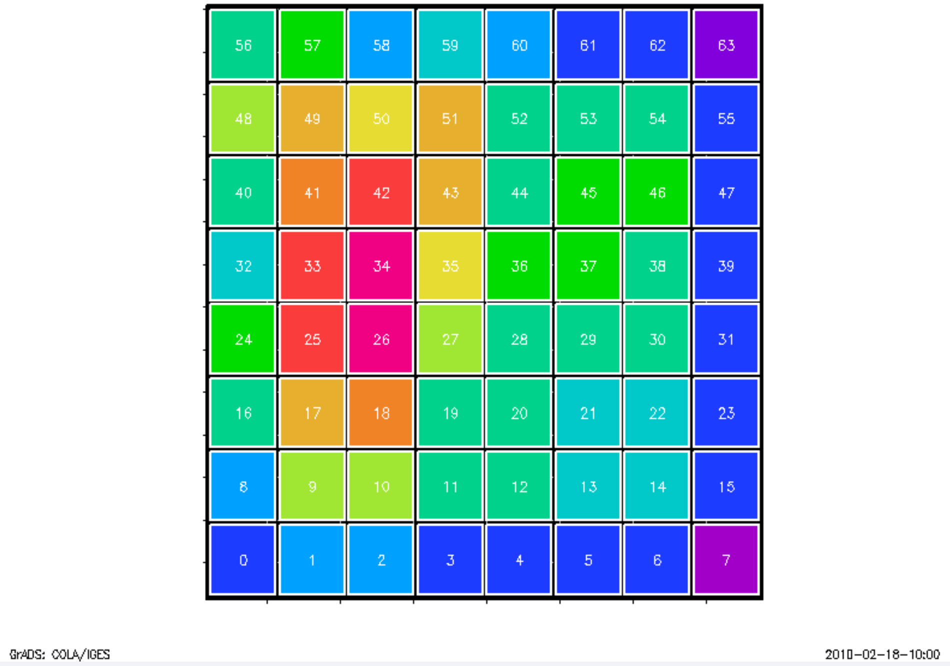


A quick Example: Weather Forecasting in BRAMS

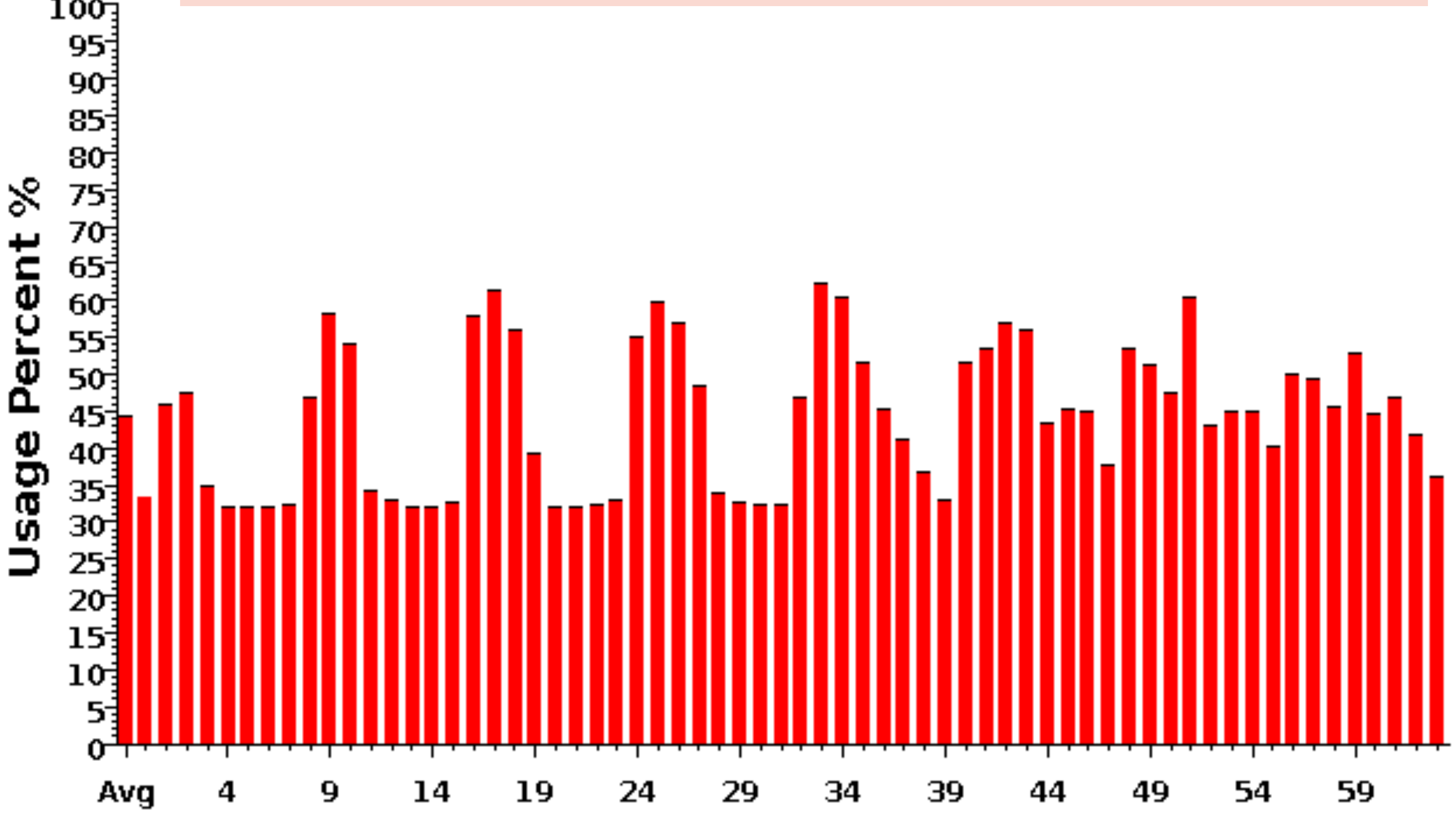
- Brams: Brazillian weather code (based on RAMS)
- AMPI version (Eduardo Rodrigues, with Mendes and J. Panetta)



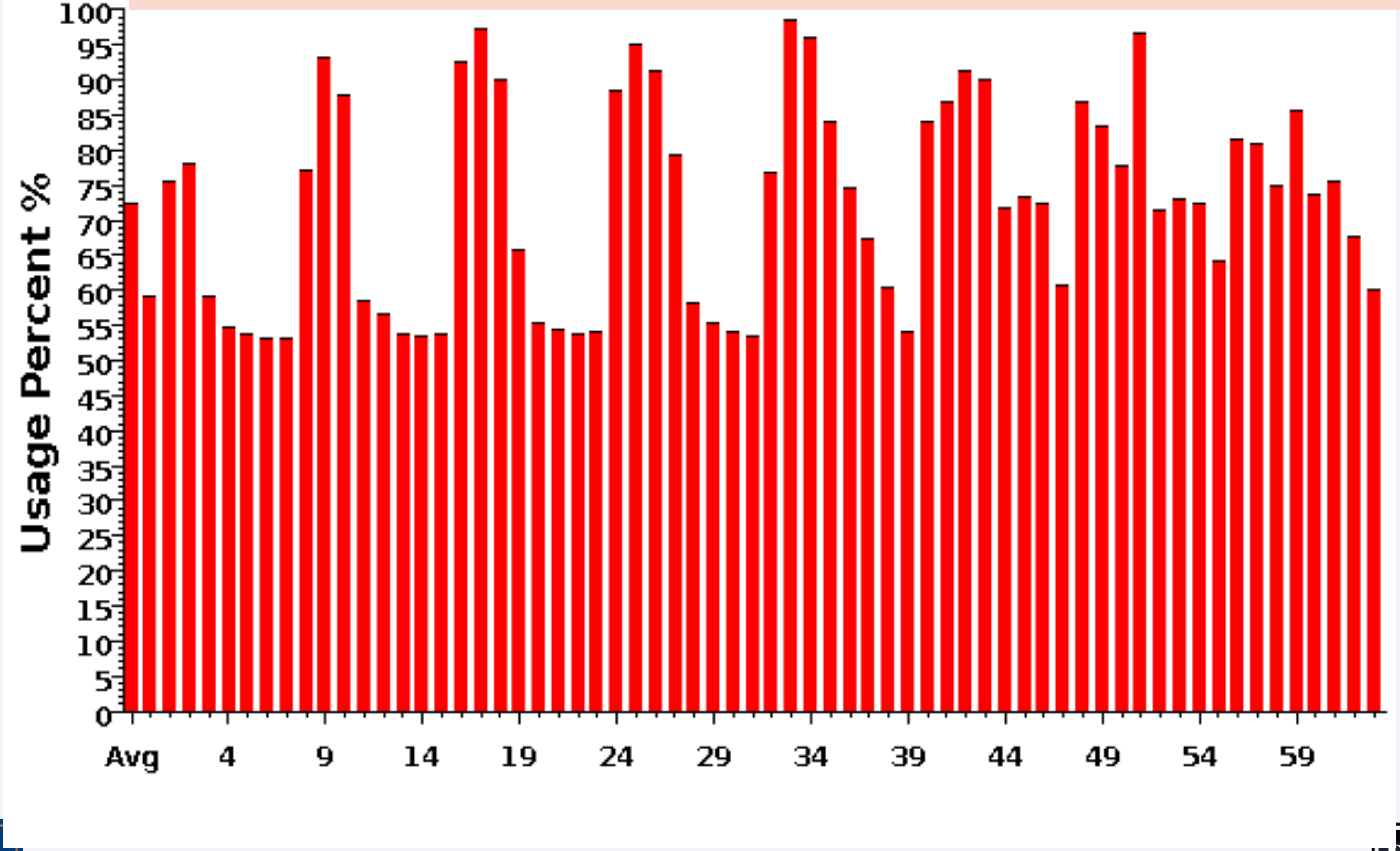

20.5S
21S
21.5S
22S
22.5S
23S
23.5S
24S
24.5S
25S
25.5S
26S
26.5S
27S
51W
50W
49W
48W
47W
46W
45W
15
13
12
11
9
8
7
6
4
3
2
0
GrADS: COLA/IGES
2010-02-18-09:46


56 57 58 59 60 61 62 63
48 49 50 51 52 53 54 55
40 41 42 43 44 45 46 47
32 33 34 35 36 37 38 39
24 25 26 27 28 29 30 31
16 17 18 19 20 21 22 23
8 9 10 11 12 13 14 15
0 1 2 3 4 5 6 7
GrADS: COLA/IGES
2010-02-18-10:00

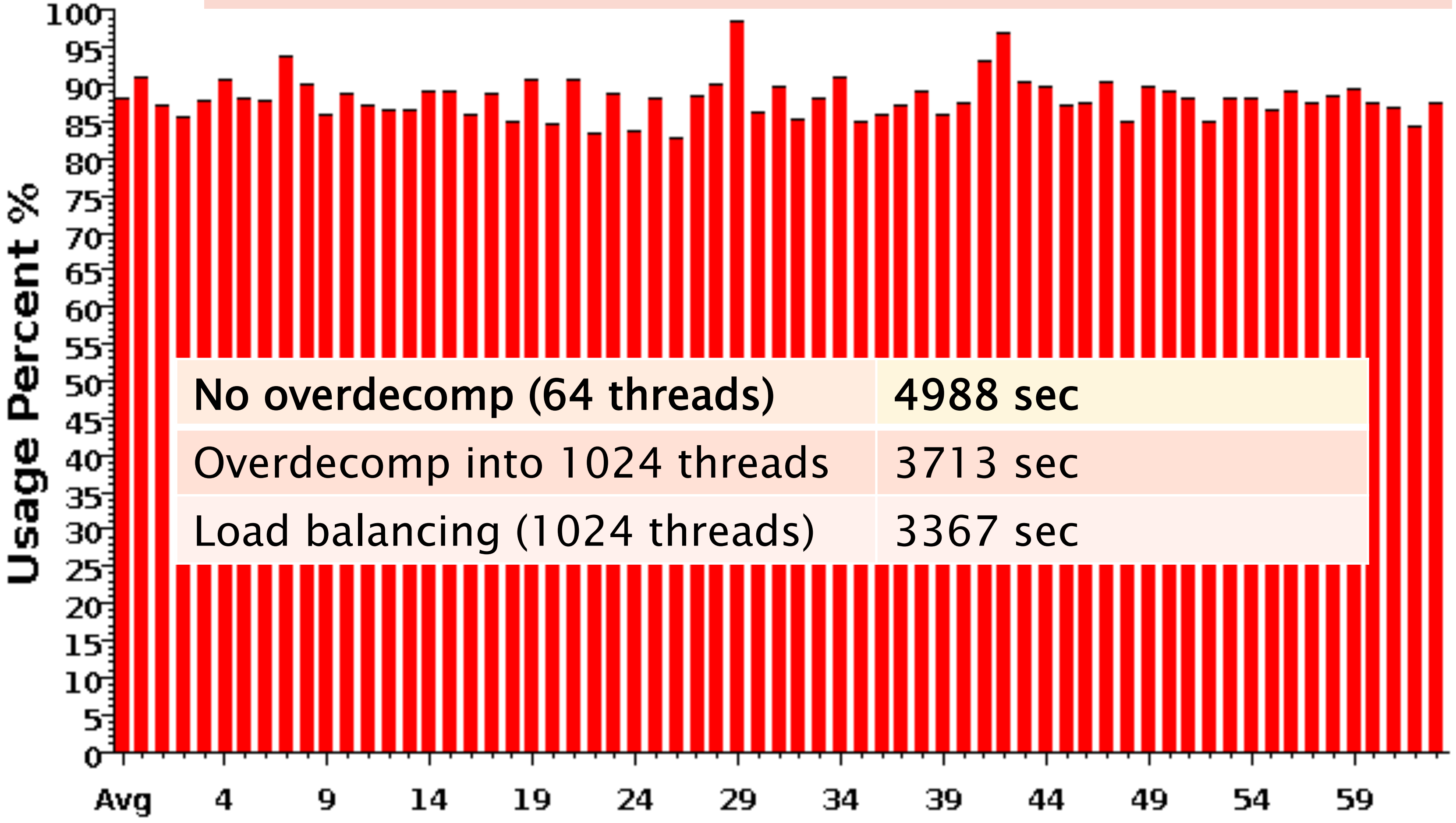
Baseline: 64 objects on 64 processors



Over-decomposition: 1024 objects on 64 processors: Benefits from communication/computation overlap



With Load Balancing: 1024 objects on 64 processors



No overdecomp (64 threads)	4988 sec
Overdecomp into 1024 threads	3713 sec
Load balancing (1024 threads)	3367 sec

Next step: world dominion!

- The world uses MPI
- AMPI provides amazing runtime-adaptivity to MPI programs
- What could go wrong?

AMPI story

- Well, there is a little step of "converting" MPI programs to AMPI
  - Mostly, just make it "thread-safe" by encapsulating global variable accesses
  - And a couple more small changes for facilitating load balancing
  - For most mid-size applications, this took an afternoon or maybe a week
  - Seemed like a worthwhile investment
- A little bit of compiler support can do this easily
- But: you need a full C/C++/Fortran compiler infrastructure to do it

Compiler support issues

- Compiler researchers;
  - Our language support needs are too simple for them
  - After all, they can deal with high-brow polyhedral stuff
  - Besides they thrive on demonstrations, rather than working systems
- Build your own infrastructure?
- Simplify language (give up on C/C++)?

Language acceptance

- An important lesson (following up from AMPI)
  - Small annoyances are big problems, if they come in the way of good initial experience
- Another Example:
  - Charm++ : mostly C++ programming, but requires an interface file describing method signatures
  - Parsing of this file is done by a simple translator
  - Not very robust, but not a problem for experienced programmers
    - As in: after your second or 3rd program, you know what works, what are the workarounds, etc.
  - But it can be a big issue for someone evaluating it afresh, and working without the benefit of experience users around them!

Interoperability

- You want modules written in your new languages to work well with modules written in existing dominant "languages"
  - E.g. MPI
- Also, interoperate with other new languages
  - Including your own other languages!
  - Because once you get the hang of it, you will be addicted designing new languages
    - Just joking
  - But we will see justification for existence and co-existence of multiple languages

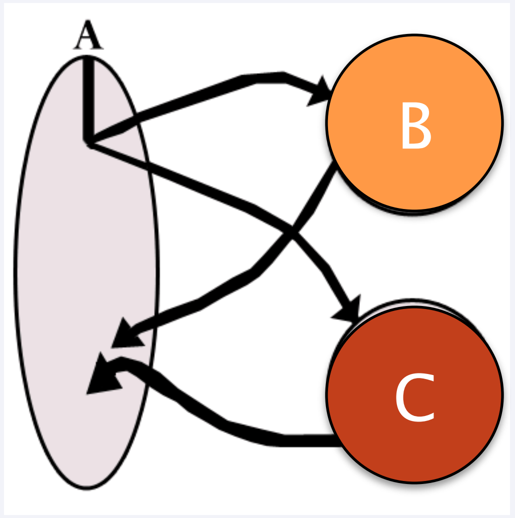
Interoperability

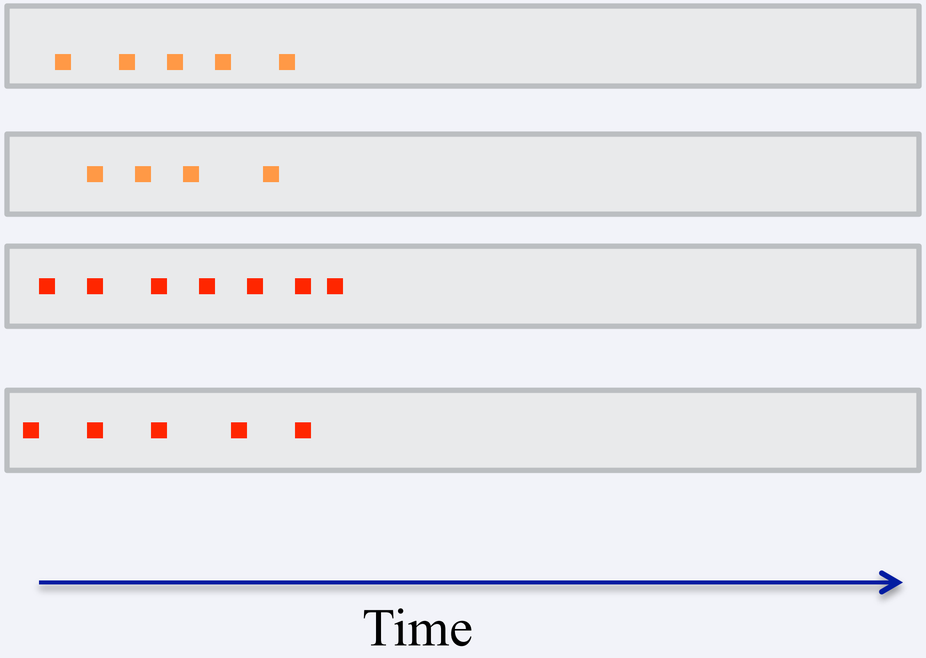
- Has multiple dimensions
- Don’t “own” the “main” and initialization
  - Every language will want to do that, and that impossible
- Don’t conflict on name-spaces
- Cannot have conflicting runtimes

Compositionality

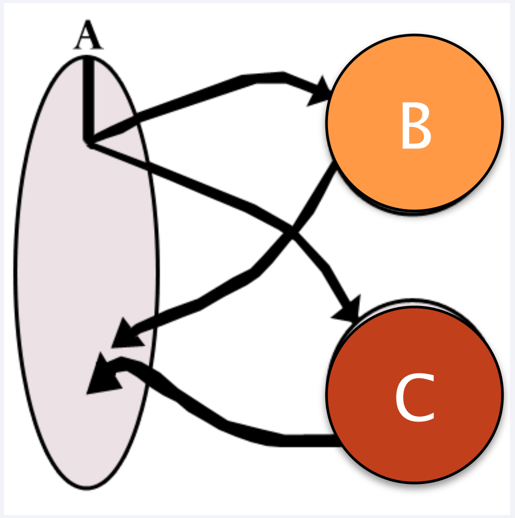
- It is important to support parallel composition
  - For multi-module, multi-physics, multi-paradigm applications…
- What I mean by parallel composition
  - B || C where B, C are independently developed modules
  - B is parallel module by itself, and so is C
  - Programmers who wrote B were unaware of C
  - No dependency between B and C
- This is not supported well by MPI
  - Developers support it by breaking abstraction boundaries
    - E.g., wildcard recvs in module A to process messages for module B
  - Nor by OpenMP implementations:

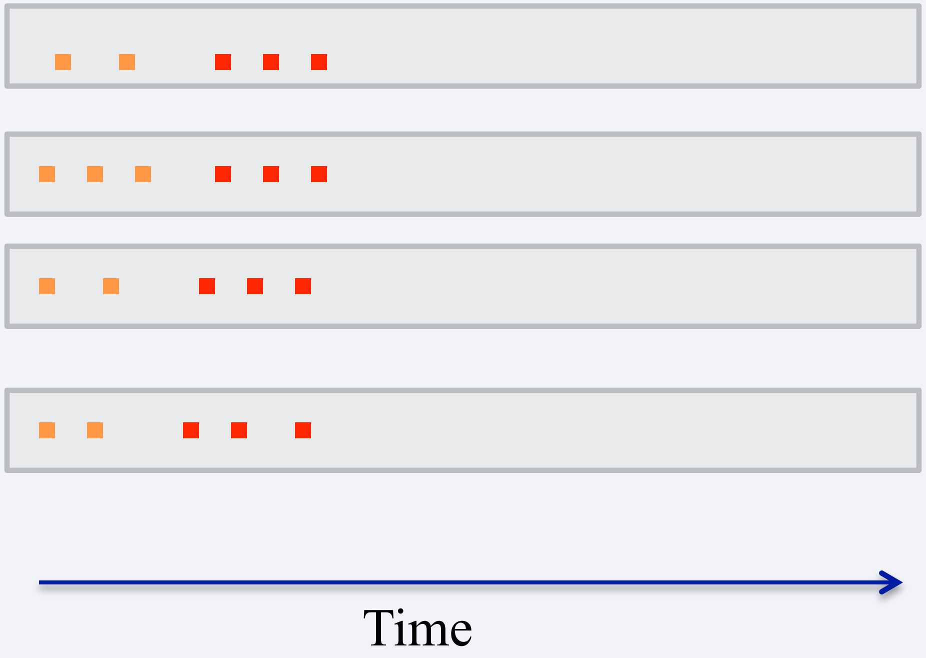
Without message-driven execution (and virtualization), you get either: Space-division



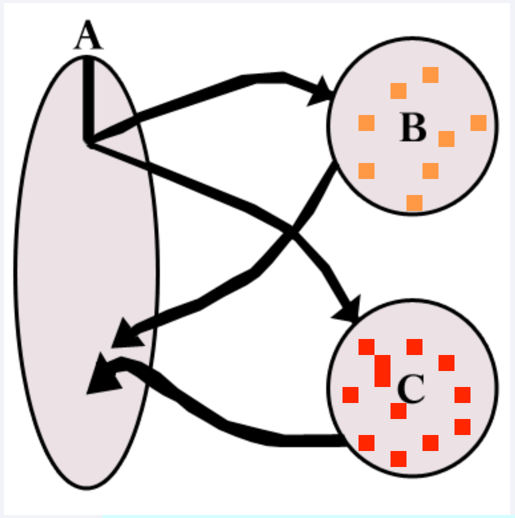


OR: Sequentialization





Parallel Composition: A1; (B || C); A2



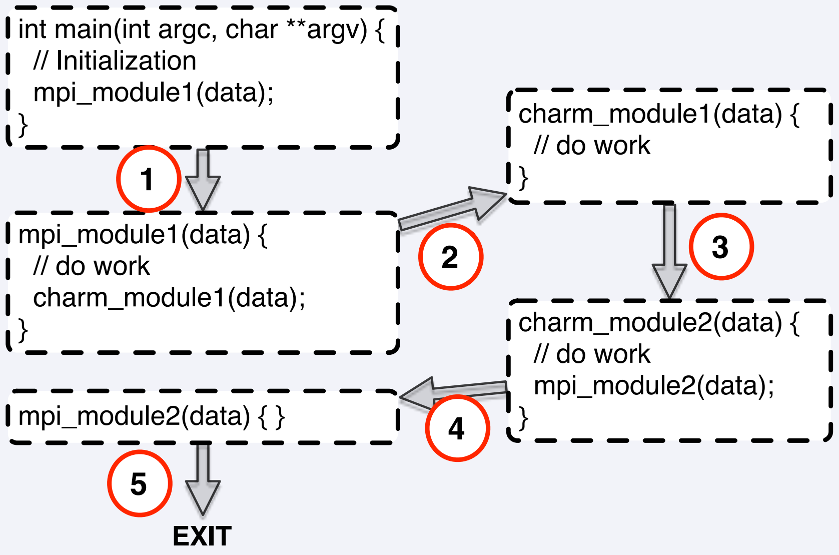
Recall: Different modules, written in different languages/paradigms, can overlap in time and on processors, without programmer having to worry about this explicitly

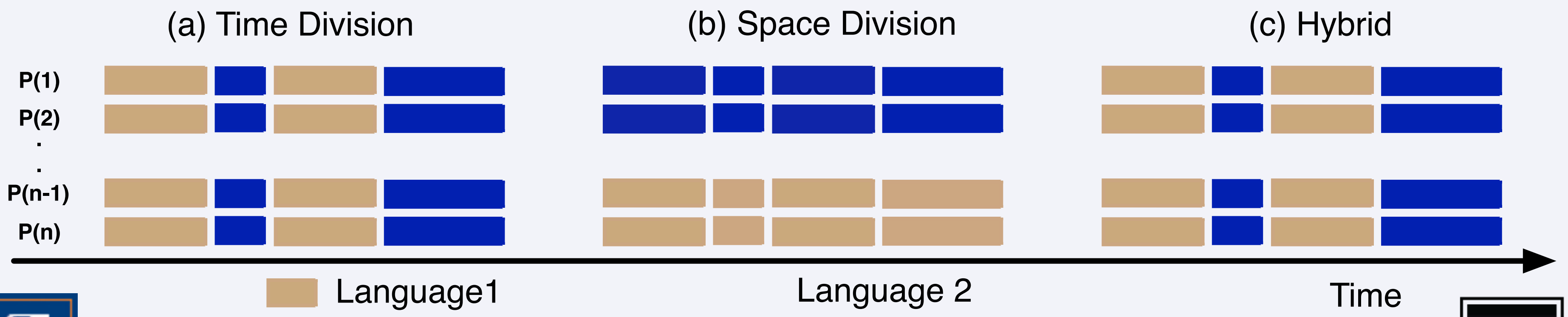
Interoperability

- Between languages with message-driven and programmer driven scheduling
  - Example: MPI and Charm++
- Essentially requires “exposing” the message driven scheduler in a library interface

Interoperation of Parallel Languages

- Implement a library in the language that suits it the most, and use them together!
- MPI + UPC, MPI + OpenMP + Charm++





Is Interoperation Feasible in Production Applications?

Application	Library	Productivity	Performance
CHARM in MPI (on Chombo)	HistSort in Charm++	195 lines removed	48x speed up in Sorting
EpiSimdemics	MPI IO	Write to single file	256x faster input
NAMD	FFTW	280 lines less	Similar performance
Charm++'s Load Balancing	ParMETIS	Parallel graph partitioning	Faster applications

High Level Programming Systems

- Different ways of attaining "higher level"
  - Global view of data
  - Global view of control
  - Both
  - Simplified or specialized syntax
  - Safety properties
- But the largest benefits come from specialization
  - Domain specific languages
  - Domain specific Frameworks
  - Interaction-pattern specific languages

Task-based languages

- Just an aside:
- Tasks used to mean "agenda" parallelism
  - Create (fire) a fully described task
  - Once created, it can run on any processor/node and has no dependences
- New definition:
  - Tasks are nodes of a computation DAG
  - They have dependences that are visible to the RTS
  - Typically run on the same node that created it

Moving Computation to Data

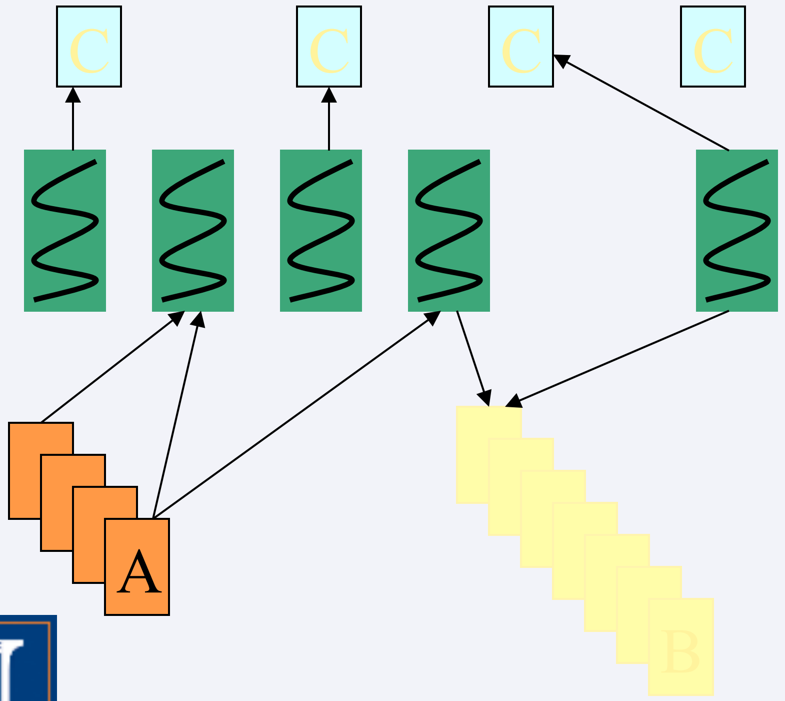
- I came across this phrase in Ann Roger’s work
- It’s a nice catchy phrase
- But really:
  - Computation is when data meets data to create data destined for other computations
  - Macro-data flow view
  - Its always data moving to data
- There is a sense in which one of the “data” is computation:
  - If it is a user-level thread, with its own stack, for example (or a continuation)

MSA: Multiphase Shared Arrays

Observations:

General shared address space abstraction is complex

Certain special cases are simple, and cover most uses

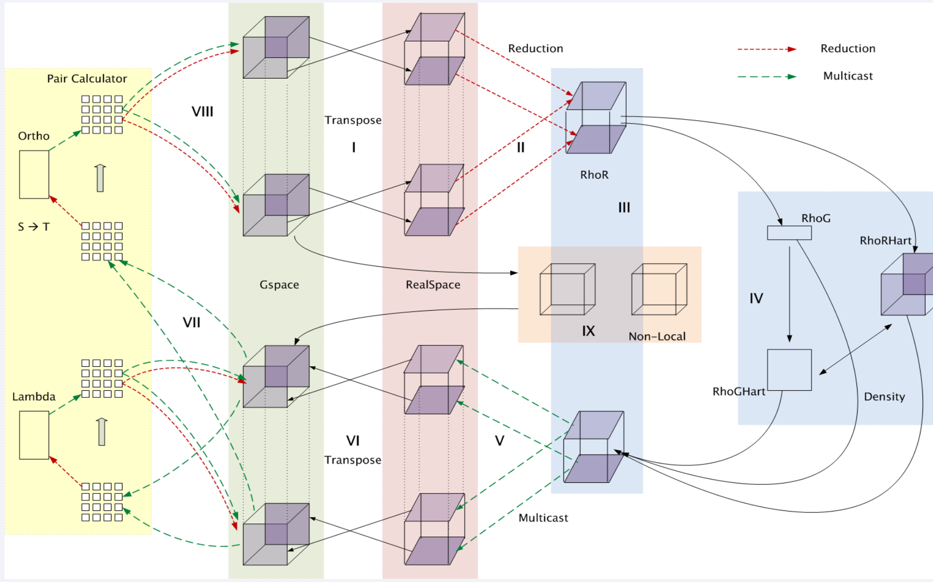


- In the simple model:
- A program consists of
  - A collection of Charm threads, and
  - Multiple collections of data-arrays
    - Partitioned into pages (user-specified)
- Each array is in one mode at a time
  - But its mode may change from phase to phase
- Modes
  - Write-once
  - Read-only
  - Accumulate
  - Owner-computes

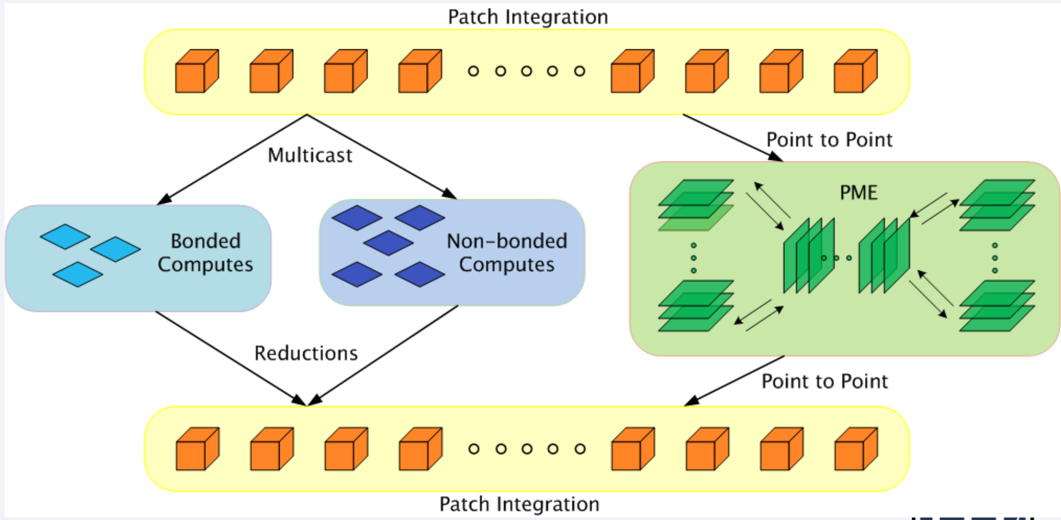
Charisma: Static Data Flow

Observation: many CSE applications or modules involve static data flow in a fixed network of entities

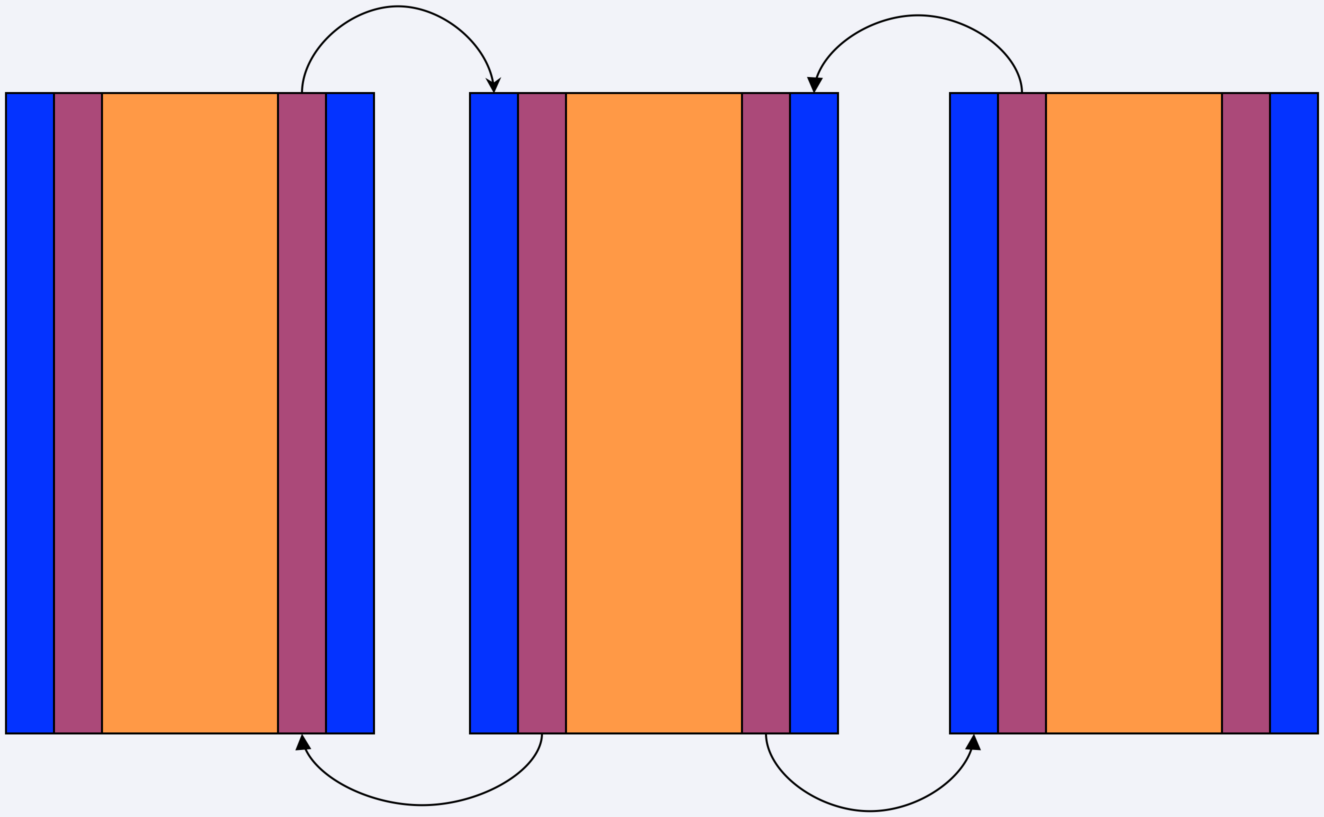
The amount of data may vary from iteration to iteration, but who talks to whom remains unchanged



- *Arrays* of objects
- Global parameter space
  - Objects read from and write into it
- Clean division between
  - Parallel (orchestration) code
  - Sequential methods



Charisma++ example (Simple)



```
while (e > threshold)
        forall i in J
          <+e, lb[i], rb[i]> :=  J[i].compute(rb[i-1],lb[i+1]);
```

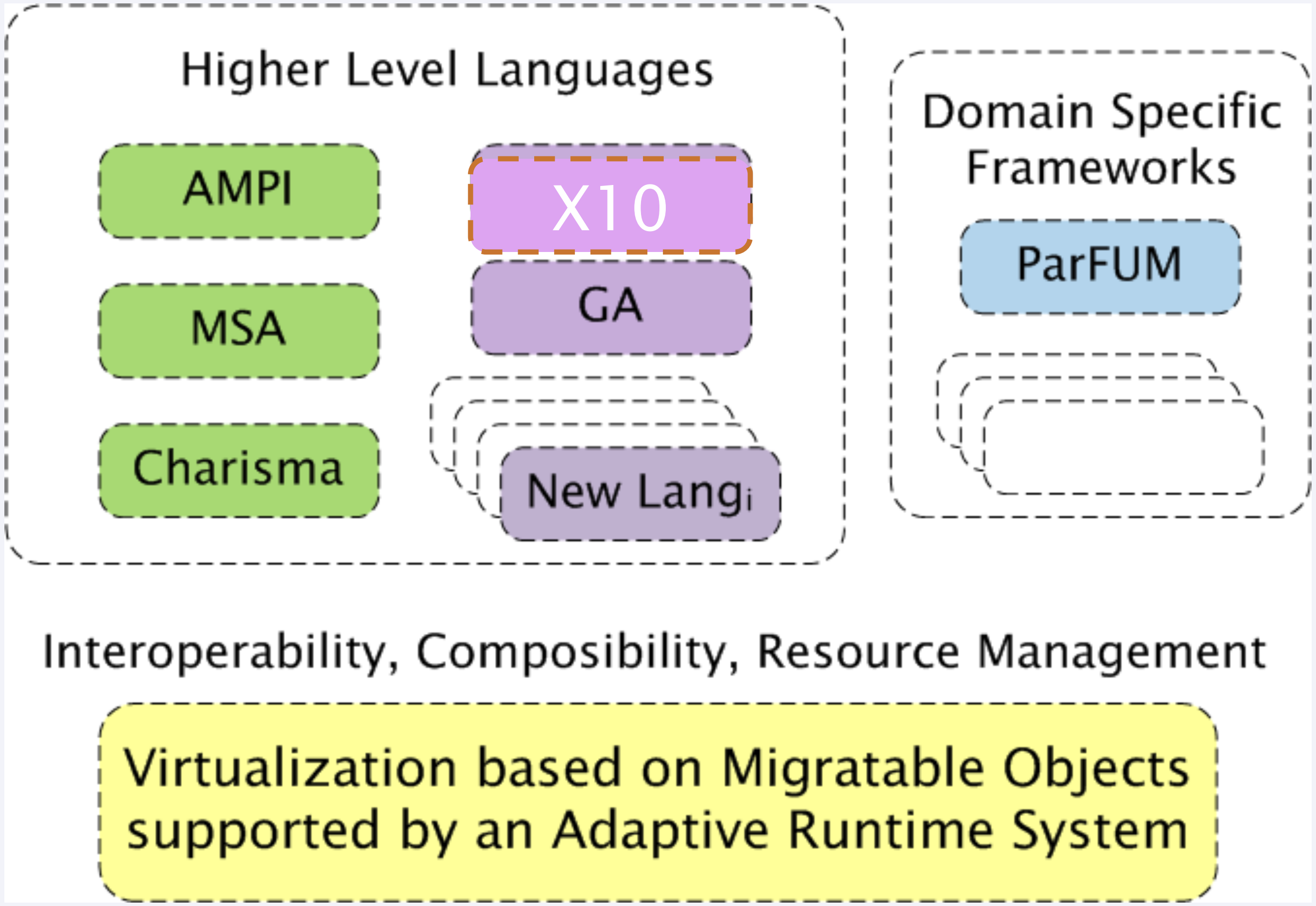
DivCon-DA

- Work in Pritish Jetley's PhD thesis
- DivCon: divide-and-conquer
- The twist: parallel arrays
- E.g. express quicksort using Divcon
  - Normal implementation will be swamped by data movement costs..
  - Permutation in every one of log P phases
- DivCon-DA supports distributed arrays
  - So, partitioning can happen in place, without data movement
  - Data movement becomes the prerogative of the RTS

DisTree

- Distributed Trees
- More of a DSL (Domain Specific language)
- Can be used to express
  - Barnes-Hut
  - Fast-Multipole
  - Smooth Particle Hydrodynamics
  - Graphics algorithms involving data stored in trees
  - …

A View of an Interoperable Future



Higher Level Languages
AMPI
MSA
Charisma
X10
GA
New Lang$_i$
Domain Specific Frameworks
ParFUM
Interoperability, Composibility, Resource Management
Virtualization based on Migratable Objects supported by an Adaptive Runtime System

Prescriptions for language design

- Aim at a good division of labor (sys/pgmr)
- Bottom up development of abstractions
- Use an overdecomposition based adaptive runtime system (and decompose accordingly)
- Application-oriented development
- Compiler support: important but tough
- Don't underestimate the "small" hurdles to acceptance
- Interoperate
- Specialization is *a* key to higher productivity
- We are heading towards an ecosystem of parallel languages

I am looking for a postdoc and/or a research programmer

More info on Charm++:
http://charm.cs.illinois.edu