Parallel Programming in the Real World!

- Andrew Brownsword, *Intel*
- Niall Dalton
- Goetz Graefe, *HP Labs*
- Russell Williams, *Adobe*

- *Moderator: Luis Ceze, UW-CSE*

Parallelism in the "Real" World

Where: Real ← {games, hpc}

Andrew Brownsword
MIC SW Architect
Intel Corporation

"Real" software is…

- Large, unwieldy, and long lived
  - Much longer & larger than intended
  - especially by the authors!
- Written by many people
  - Of widely ranged skills, styles, agendas, experiences
  - Many have moved on to `[next_task..next_life]`
- Hard to change
  - Even with language-aware tools

“Real” software is…

- Expressed in *and defined by* programming models
- Many levels of abstraction, concreteness, explicitness, constraints
  - Best balance depends on goals & *changes over time*
- Compilers only understand the programming model*…*
  - *…not the abstractions built in it*

"Real" parallel software is…

- Supposed to be fast
  - But performance is *extremely fragile*
- Supposed to be robust
  - Data races, deadlocks, livelocks, etc
  - Composability
- Supposed to be maintainable
  - Critical aspects often hidden in the details

1. Brief survey of the “Real” landscape
2. Implications for programming models

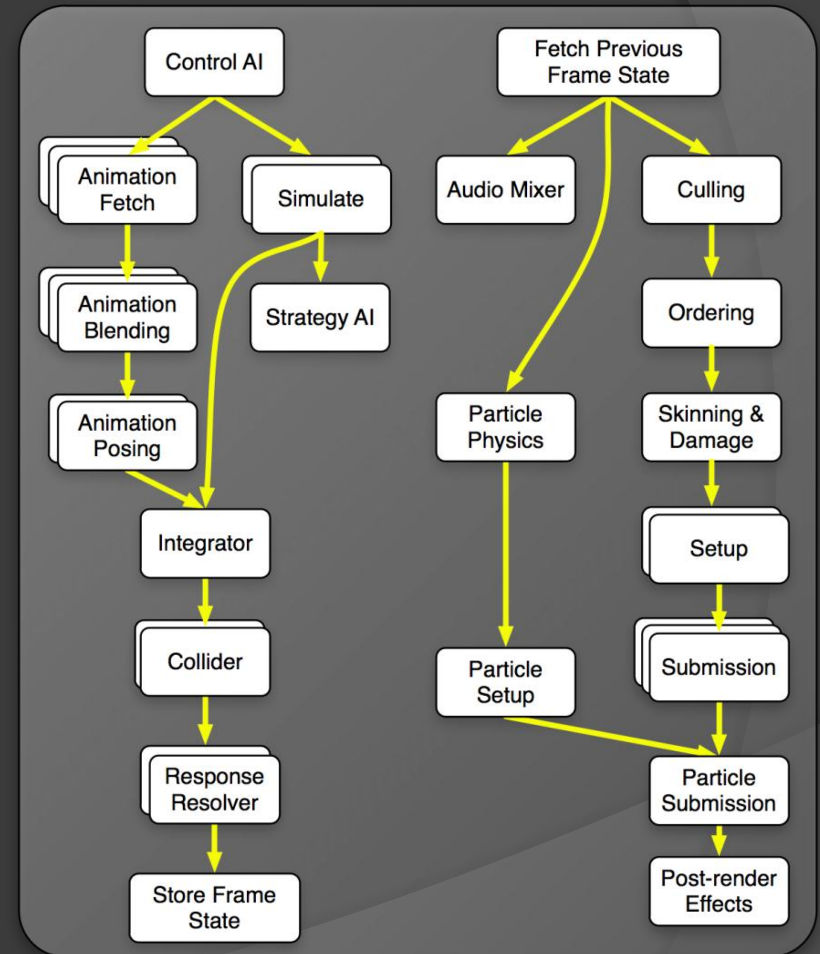
Games

Games – code & platform

- Medium-to-large codebases
  - 50K-5M lines of code, largely C++
  - May have large tool chains & online infrastructure
- *Many* diverse sub-systems running at once
  - "Soft" real-time, broad range of data set sizes
  - Frame-oriented scheduling (mostly)
  - Many sequencing dependencies between tasks
- Target hardware "is what it is"
  - Phones to servers, performance is critical
  - Multi-core, heterogeneous, SIMD, GPU, networks

Games – many systems

- Graphics
- Environment
- Audio
- Animation
- Game logic
- AI & scripting
- Physics
- User Interface
- Inputs
- Network
- I/O (streaming)
- Data conversion / processing



Note that this slide is a *gross* over-simplification!

Games – dev process

- Short development cycles
  - Severe code churn, high pressure to deliver quickly
  - Middleware & game “engine” use common
- Rapidly changing feature requirements
  - Fast iteration during development is critical
  - Code & architecture maintenance nightmare
- Substantial volume & variety of media data
  - Content team size greatly exceeds engineering’s
- Porting between diverse platforms is common

HPC

HPC – code & platform

- Widely varying codebases & domains
  - 10K-10M+ lines of code
  - Diverse programming models
    - Fortran, C/C++, MPI, OpenMP dominate
- Few kernels*, **BIG** data*
  - Correctness & robustness are critical*
  - Epic, titanic, gargantuan data sets*
- Varied target hardware
  - Workstations to large clusters
  - Often purchased for the application

* Usually

HPC – dev process

- On-going development cycles
  - Code is generally never *re*-written, lasts for decades
  - Huge, poorly understood legacy code
  - Heavy library use (math, solvers, communications, etc.)
- Correctness, verifiability, robustness
  - Dependability of results is crucial
  - Very, very long running times are common
- Portability across generations & platforms
  - Tuning 'knobs' exposed rather than changing the code
  - Outlast HW, tools, vendors, prog. models, authors

Programming Models

Programming Models

- Design of the model has formative impact on software written in it
  - Abstractions to avoid over-specifying details
  - Concrete to allow control over solution
  - Explicit to keep critical detail visible
  - Constraints to allow effective optimization
- For parallelism:
  - Top desirable attributes
  - Top factors to address

Desirable Attributes…

Integration

- With other models:
  - Existing model, enables *gradual adoption*
  - Peer models, there is no *single silver bullet*
  - Layered models, enables DSLs & interop
- With runtimes:
  - Interaction & interop within processes
  - Resource management (processors, memory, I/O)
- With tools:
  - Build systems, analysis tools, design tools
  - Debuggers, profilers, etc.

Portability

- Hardware, OS & vendor independence
- Standard, portable models live longer
- Investment in software is very expensive
  - Re-writing is often simply not an option
  - Even seemingly small changes can be extraordinarily expensive
    - Testing & validation costs
    - Architectural implications

Composability

- Real software is large and complex
  - Built by many people
  - Built out of components
  - Subject to intricate *system level* behaviours

- Programming models must facilitate and support these aspects

Factors To Address…

Concurrent Execution

- Multiple levels to achieve performance
  - Vectorization (SIMD & throughput optimization)
  - Parallelization (multi/many-core)
  - Distributed (cluster-level)
  - Internet (loosely coupled, client/server, cloud services)

- Programming model needs to express each level
  - *Each* level brings >10x potential
  - Cannot afford a different decomposition at each level

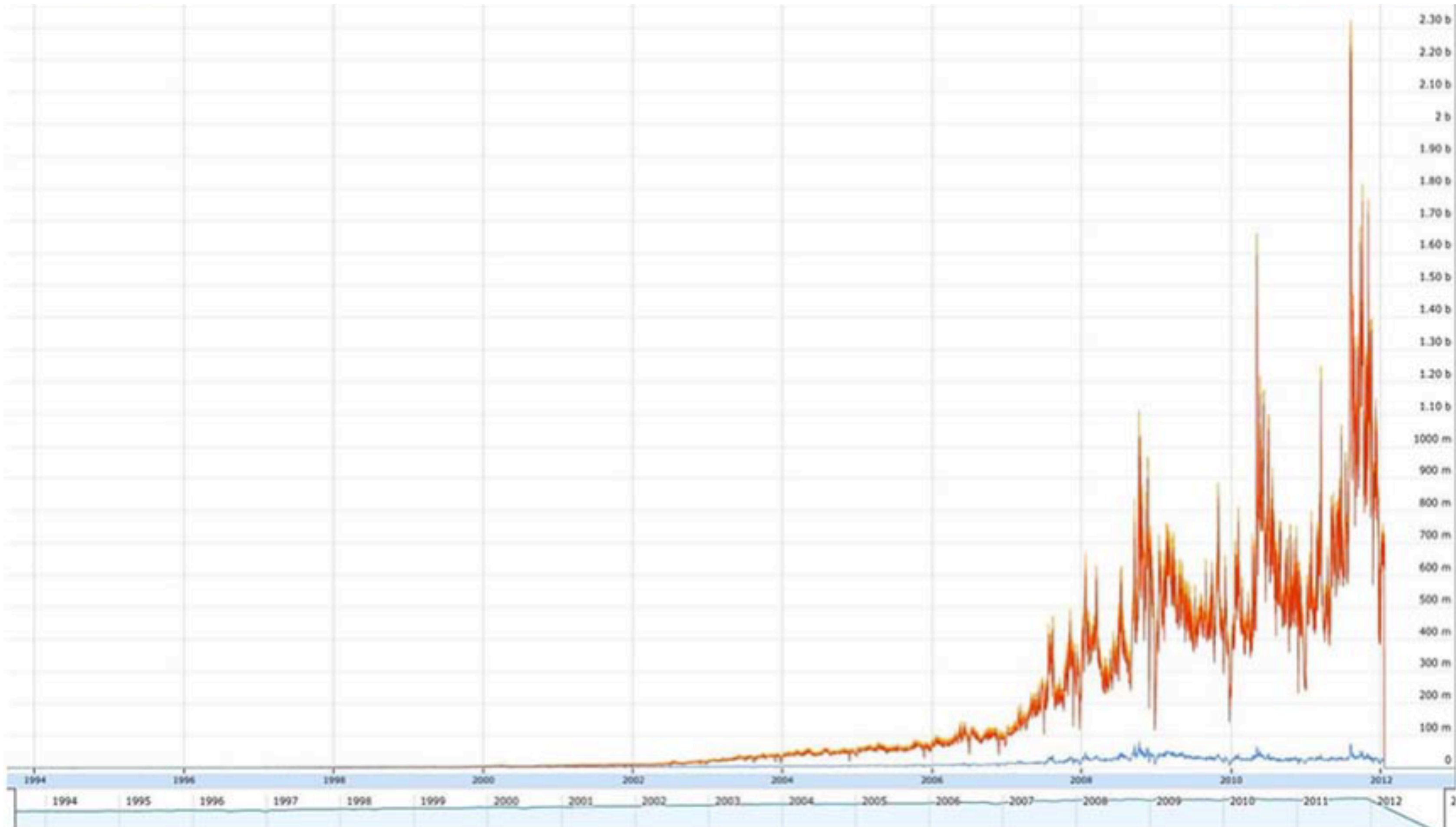
Data Organization & Access

- FLOPS are cheap, bandwidth is not
  - Severe and worsening imbalance
  - No sign of this changing
- Optimizing data access is usually key to achieving performance & power
- Existing models do very little to address this
  - Access patterns usually implicit, layouts explicit
  - Changing data layout requires changing code
  - Different hardware, algorithms & models demand different layouts

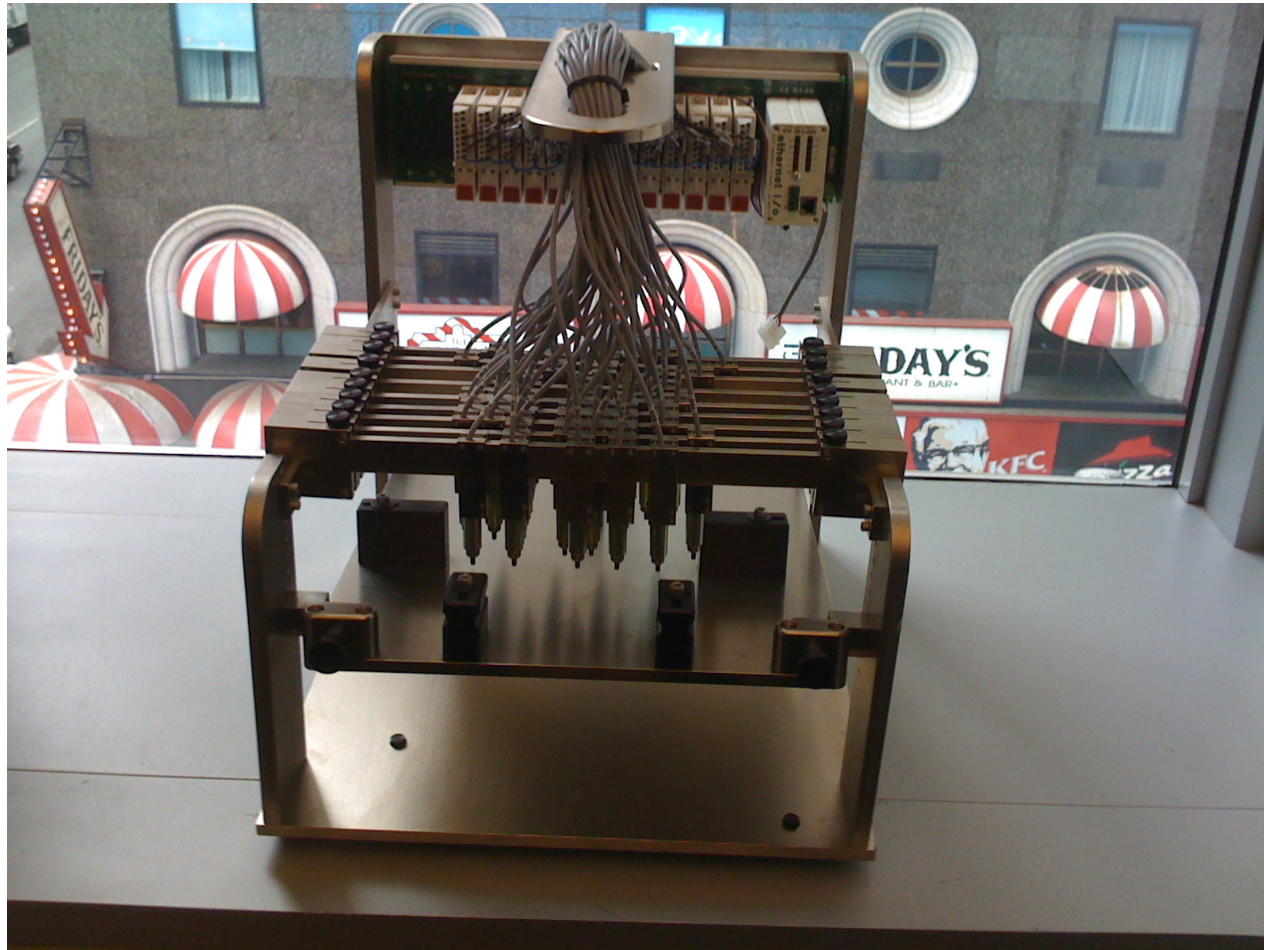
Specialization

- Hardware is diversifying
  - Heterogeneous processors (CPU, GPU, etc)
  - Fixed function hardware
  - System-on-chip
- Driven by power & performance
- Tight integration needed for fine-grained interactions & data

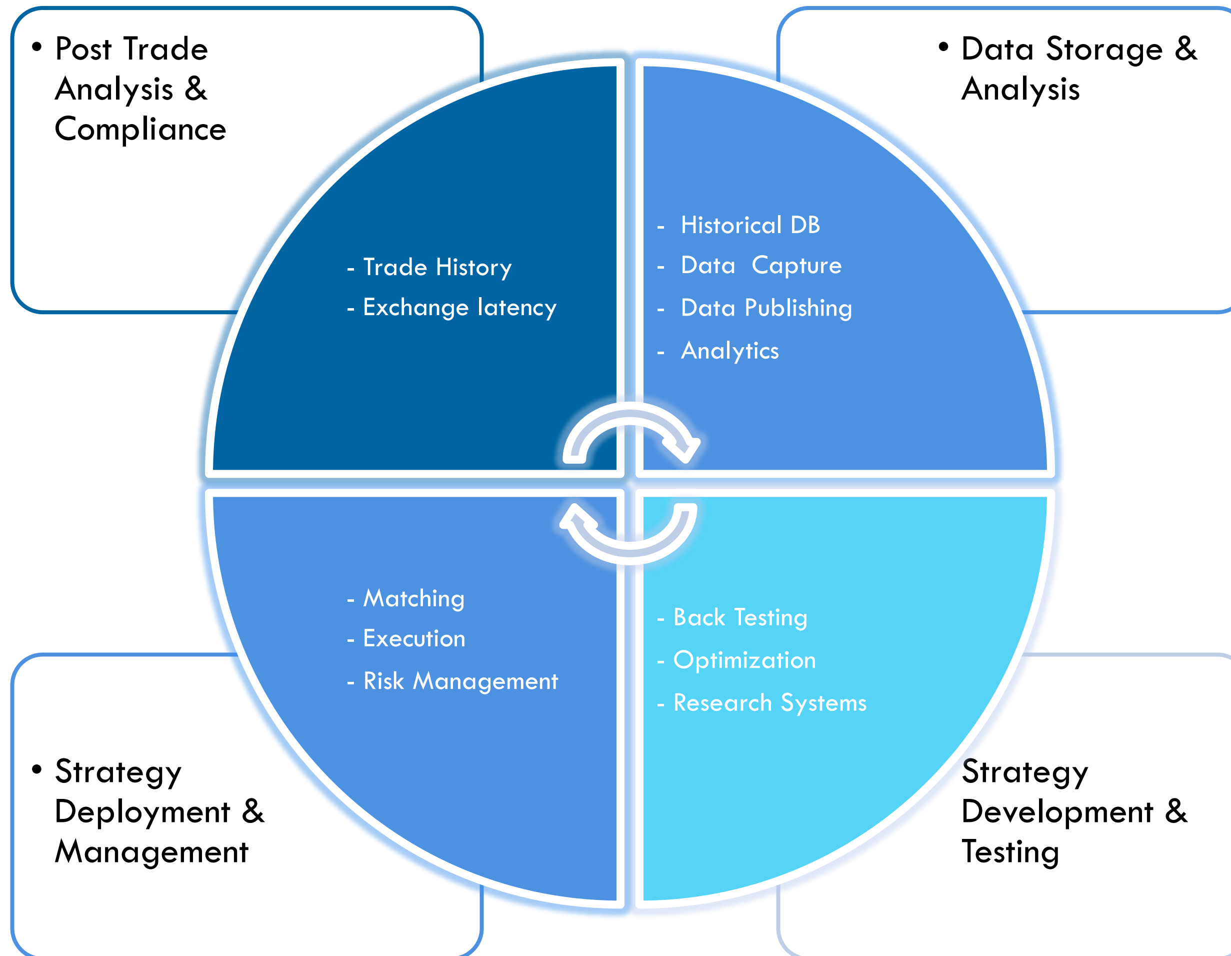
NYSE TAQ record counts



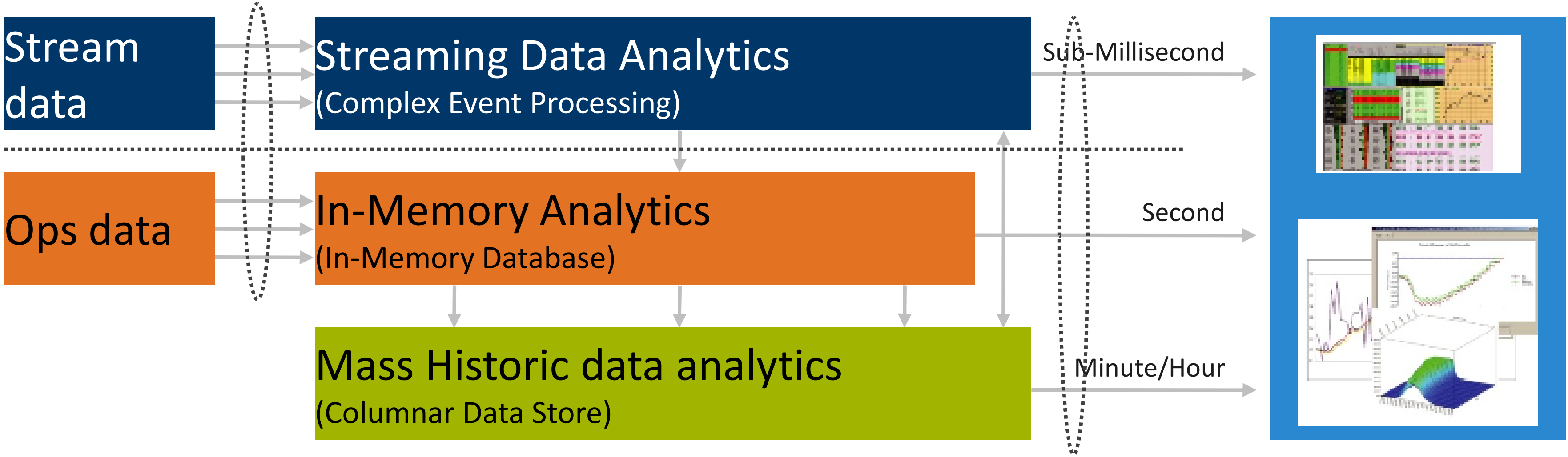
An elegant weapon..
for a more civilized age



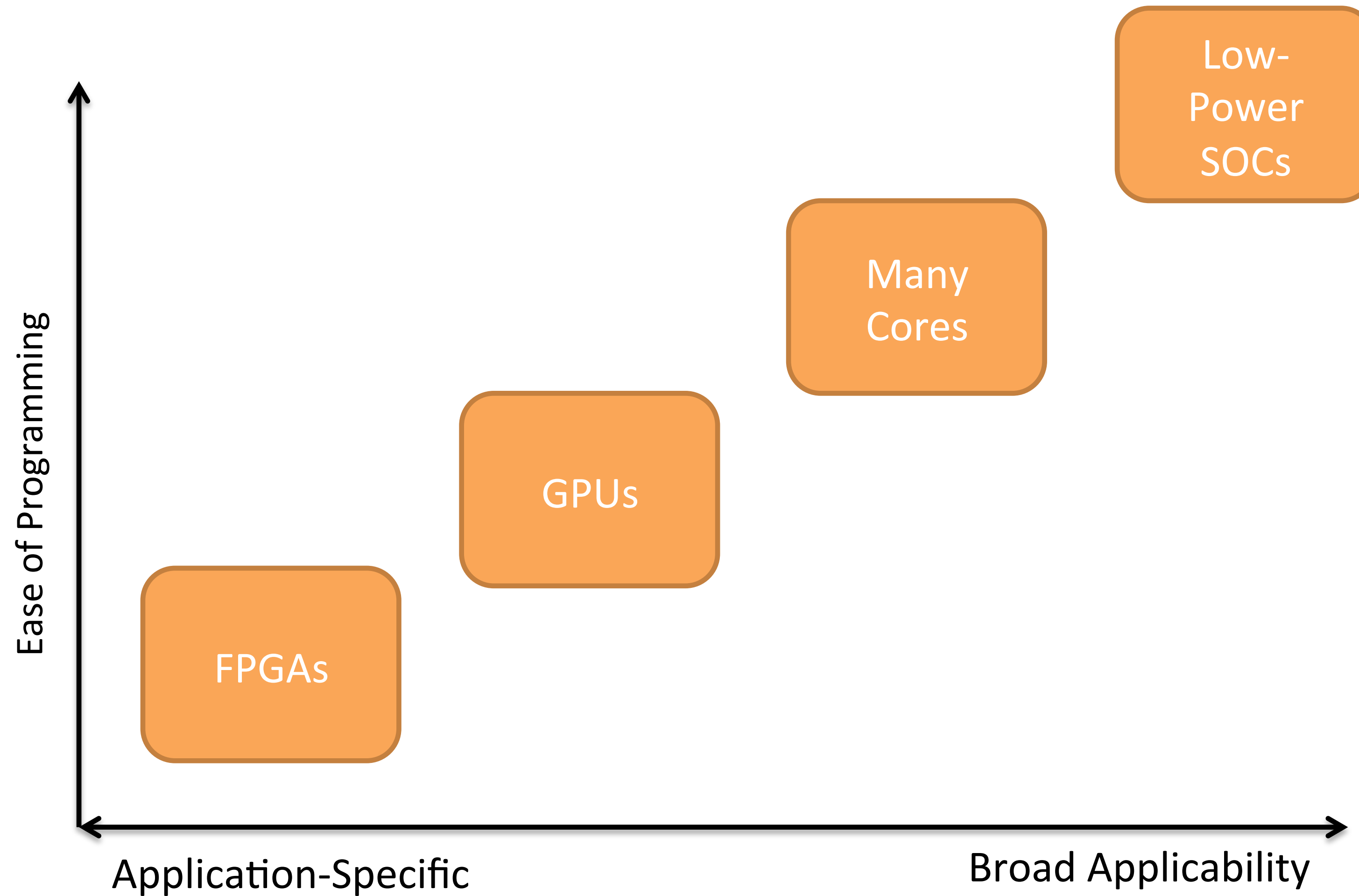
Trade lifecycle



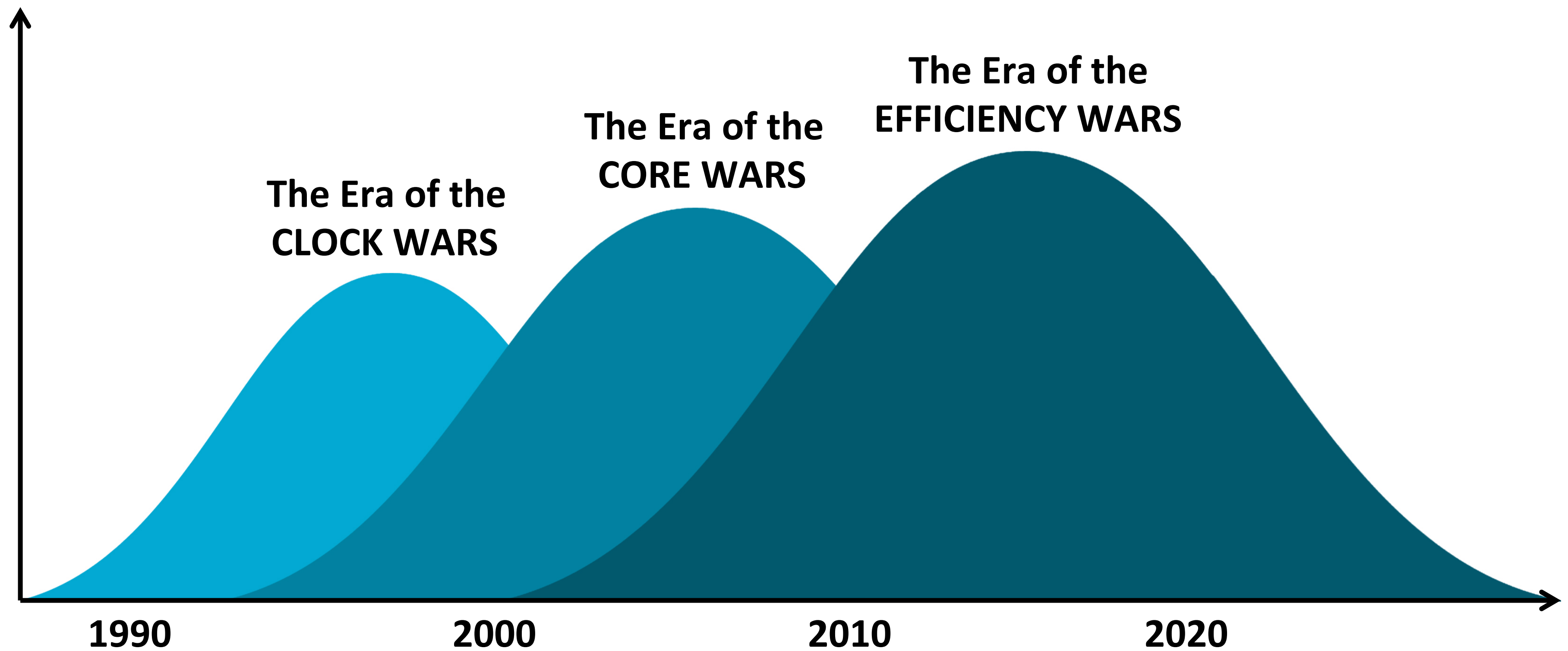
Follow the data



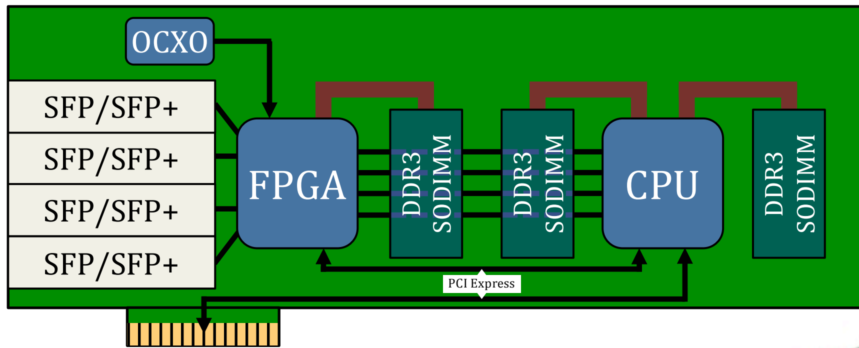
Different cores for different chores



Don’t fight the last war



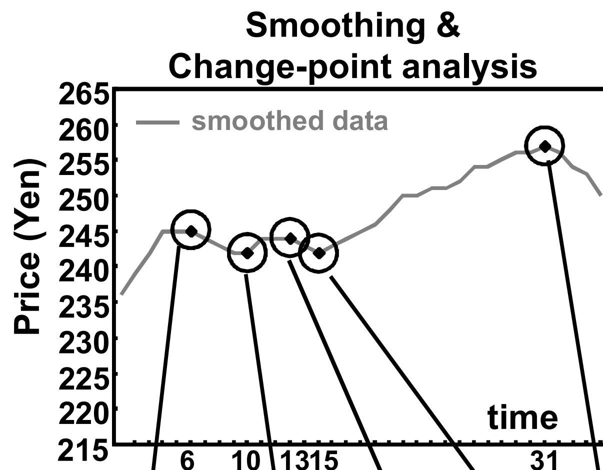
Softer HW or harder SW?

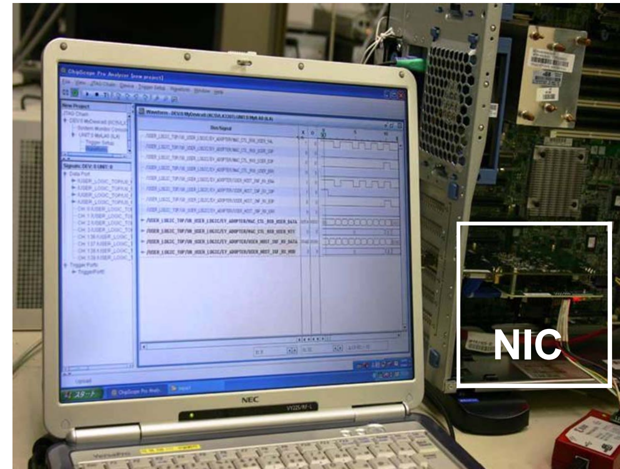


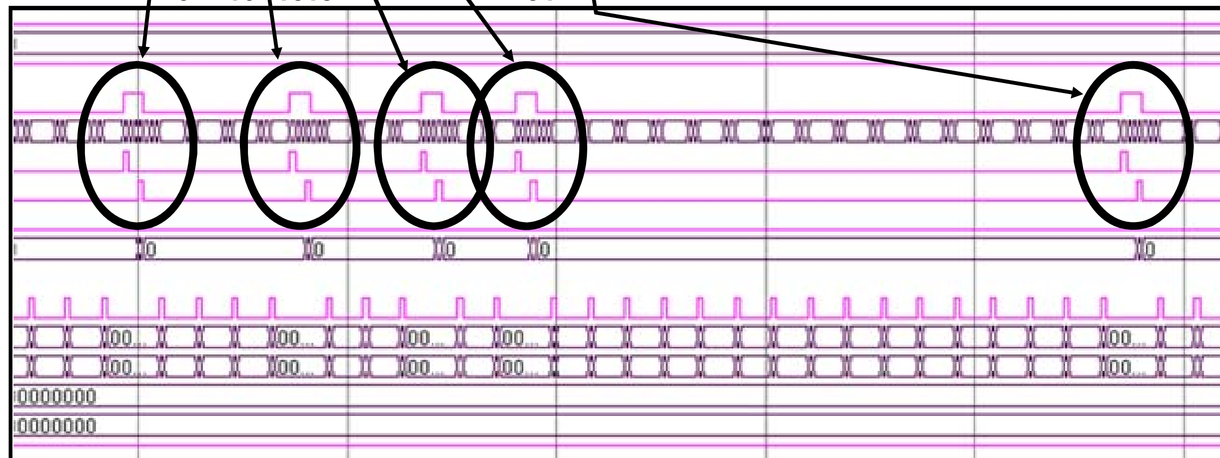
PCI Express Card comprising:

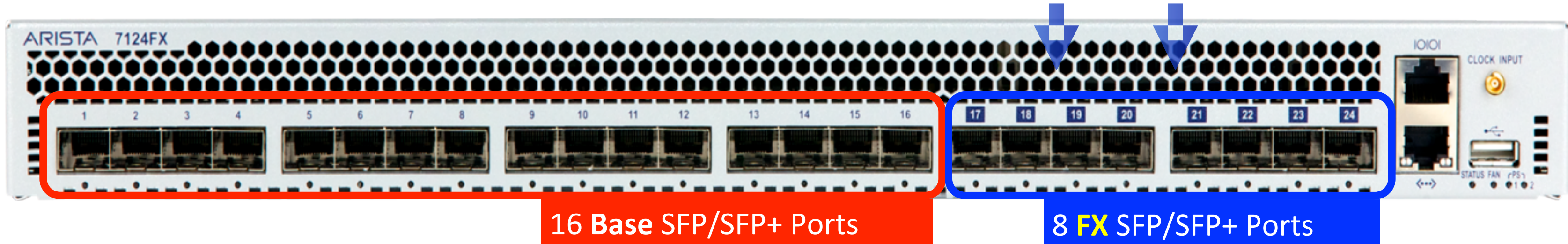
- 4 SFP/SFP+ cages providing flexible 1/10Gbps connectivity
- Tilera TILE-Gx8036™ Processor (64-bit, 36 cores, 1.2GHz) - two dedicated 1600MHz DDR3 SODIMMs
- Altera® Stratix® IV FPGA - 531,200 Eq. LEs – One dedicated DDR3 SODIMM
- High-precision (OCXO) Oven-compensated Quartz Oscillator providing ultra-accurate timestamping capability
- Generation 2 (5Gbps) x8 PCI Express bus providing 40Gbps between card and host

Data reduction









SRAM
SRAM
DRAM
FPGA
Custom Switch ASIC
CPU
DRAM
SSD
FRONT PANEL PORTS

```
somedata = (1,2,3); otherdata = [1,2,3]; dict = [`a=1, `b=2]; // Note these are different types, list vs. vector.
type area = `FX | `Equities Int | `FixedIncome Double Double

results@node0 with f = #(id :: symbol; profit :: double)

f = {(id, area, pnl)
  var profit = area? `FX : pnl*.98 | `Equities x : pnl-x |  `FixedIncome x y : pnl*(x-y);
  insert (id, profit) into results
}

jobs = select id, area, parameters from strategies where date==today()

simulate = {(job)
  var pnl = sum(random * 1..10);
  if (pnl > 100) {send (job.id, job. area, pnl) to results; (`ok, pnl)}
  else (`fail, pnl)
}

job_status = @[select distinct processors from places] {
 <-[(x){begin simulate(x)} each jobs]
}

failed_jobs = select (status, pnl) from job_status where status==`fail
```

```
// run some code in place A; block until it's done
@A {code}

// start an activity f in place B and return immediately
@B begin {f}

// run some code in place C, taking ownership of data
@c with data {…} // bind data to place

// distribute data over place1 and place 2
@[place1, place2] data;

// redundant copies of data in place1 and place 2; also works for redundant computation
@[place1], [place2] data;

// Run f in the fastest place we can
var c = select core-id from processors where max frequency
@c {f(`somedata)}

// Queue work in parent
@parent {code}

// Reply
@reply {code}
```

Open Problems

- Machines are already beyond our ability to program productively with high performance
- It's getting harder to observe, understand, debug & tune our broken programs/machines
- Where is the inconsistency coming from?
- What implicit effects are we suffering from?
- How do we cope with increasing diversity?
- What do we need to give up to get some help?

Hot topics in parallelism in data management

Goetz Graefe
Hewlett-Packard Laboratories
Palo Alto, Cal. – Madison, Wis.

History

- Concurrency among independent transactions
  - Each transaction single-threaded
  - 1960s, 1970s, …
- Parallel query processing (within a transaction)
  - Teradata 1983-84 specialized hardware
  - Gamma 1984-88 off-the-shelf hardware
  - Pipelines for algebraic execution
  - Partitioning intermediate results

Transactions

ACID = atomicity, consistency, isolation, durability

- User transactions
  - Database contents queries & updates
  - Locks held to transaction commit
  - Rollback using recovery log
- System transactions
  - Database representation changes, e.g., B-tree node split
  - In-memory data structures, “latches”

Two types of transactions

	User transactions	**System transactions**
Invocation source	User request	System-internal
Database effects	Logical database contents	Physical database representation
Data location	Database or buffer pool	In-memory page images
Invocation overhead	New thread	Same thread
Locks	Acquire & retain	Test for conflicts
Commit overhead	Force log to stable storage	No forcing
Logging	Full "redo" & "undo"	"Redo" only usually
Failure recovery	Rollback	Completion
Hardware opportunity	Non-volatile memory	Transactional memory

Two types of concurrency control

	Locks	**Latches**
Separate…	User transactions	Threads
Protect…	Database contents	In-memory data structures
During…	Entire transactions	Critical sections
Modes:	Shared, exclusive, update, intention, escrow, schema	Shared, exclusive
Deadlock…	Detection & resolution	Avoidance
… by…	Waits-for graph analysis, timeout, transaction abort, partial rollback, lock de-escalation	Coding discipline, lock leveling
Kept in…	Lock manager's hash table	Protected data structure

Current trends and challenges

- Scalability
  - Query processing versus map-reduce (Hadoop etc.)
  - Data mining, business intelligence, analytics
  - Utilities (load, reorganization, …)
- Implementation techniques
  - Low-level synchronization
  - Transactional memory
  - Non-volatile memory
  - Other novel hardware

Me: Russell Williams. My product: Photoshop

- Huge cross-platform code base on single threaded framework
- Parallel computation since mid-90s using basic parallel_for
- Scaling falls off beyond 4 cores for many operations.
- Must trade off throughput for latency
- Proliferation of thread pools

© 2011 Adobe Systems Incorporated. All Rights Reserved.

Adobe

Challenges — structure of the problem

- Asynchrony vs. parallel compute
- Available parallelism
  - Amdahl's law vs. events, views, PCI bus
  - On server, parallelize per user. On desktop: one user
  - Bandwidth limited — FLOPS / memory reference
- 80-core chips not coming; software can't use 'em.

© 2011 Adobe Systems Incorporated. All Rights Reserved.

Adobe

Challenges — structure of the solutions

- Heterogeneous environment
  - C machine vs. data parallel, high latency, high throughput
  - Different cache / memory hierarchies

- Rapidly changing hardware landscape
  - Discrete->Integrated GPU
  - SSE -> AVX -> AVX2 -> AVX3
    - __m128i vDst = _mm_cvttps_epi32(_mm_mul_ps(_mm_cvtepi32_ps (vSum0), vInvArea));

- Variety, rapid evolution, and fragmentation of tools
  - CUDA / DX / OpenCL / C++ AMP, vectorizing compilers

© 2011 Adobe Systems Incorporated. All Rights Reserved.

Adobe

Desktop GFlops (8-core 3.5GHz Sandy Bridge + AMD 6950)