EECS 262a
Advanced Topics in Computer Systems
Lecture 13

Resource allocation: Lithe/DRF
March 7th, 2016

John Kubiatowicz
Electrical Engineering and Computer Sciences
University of California, Berkeley

http://www.eecs.berkeley.edu/~kubitron/cs262

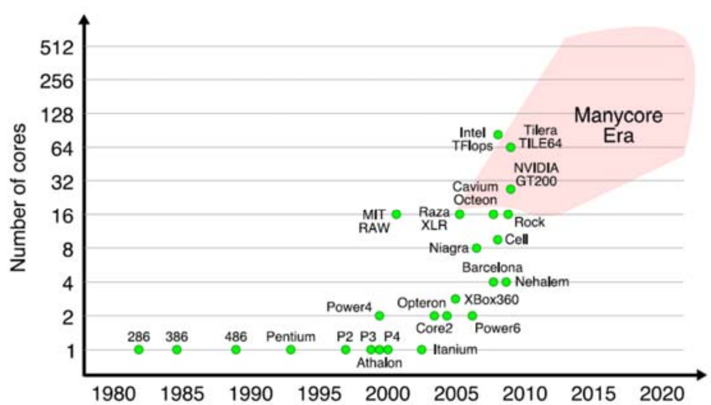
Today's Papers

- **Composing Parallel Software Efficiently with Lithe**
  Heidi Pan, Benjamin Hindman, Krste Asanovic. Appears in Conference on Programming Languages Design and Implementation (PLDI), 2010
- **Dominant Resource Fairness: Fair Allocation of Multiple Resources Types**,
  A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, Usenix NSDI 2011, Boston, MA, March 2011

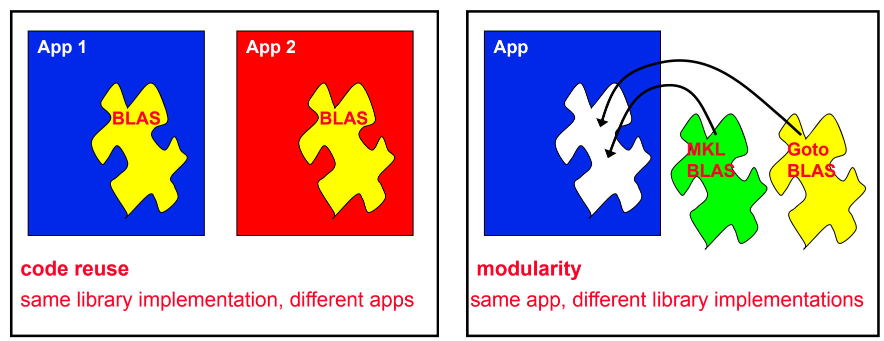
- Thoughts?

The Future is Parallel Software

Challenge: How to build many different large parallel apps that run well?
- *Can't rely solely on compiler/hardware:* limited parallelism & energy efficiency
- *Can't rely solely on hand-tuning:* limited programmer productivity



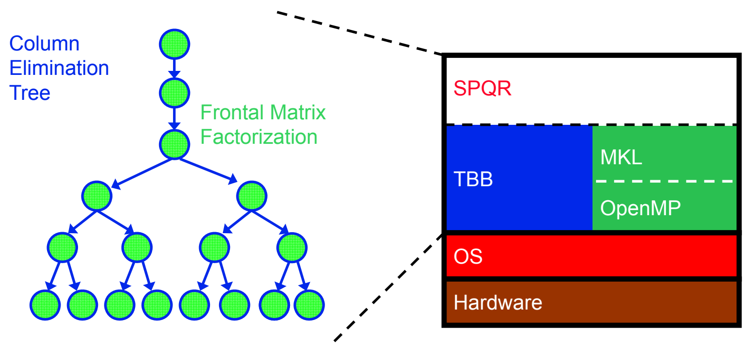
Composability is Essential



Composability is key to building large, complex apps.

Motivational Example

Sparse QR Factorization
(Tim Davis, Univ of Florida)



Software Architecture

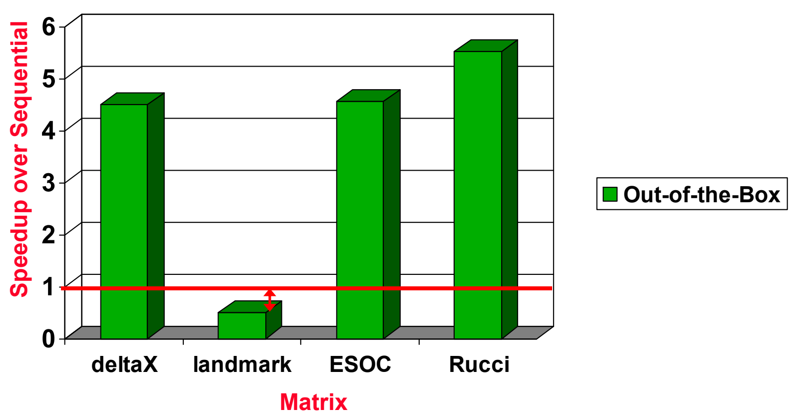
System Stack

TBB, MKL, OpenMP

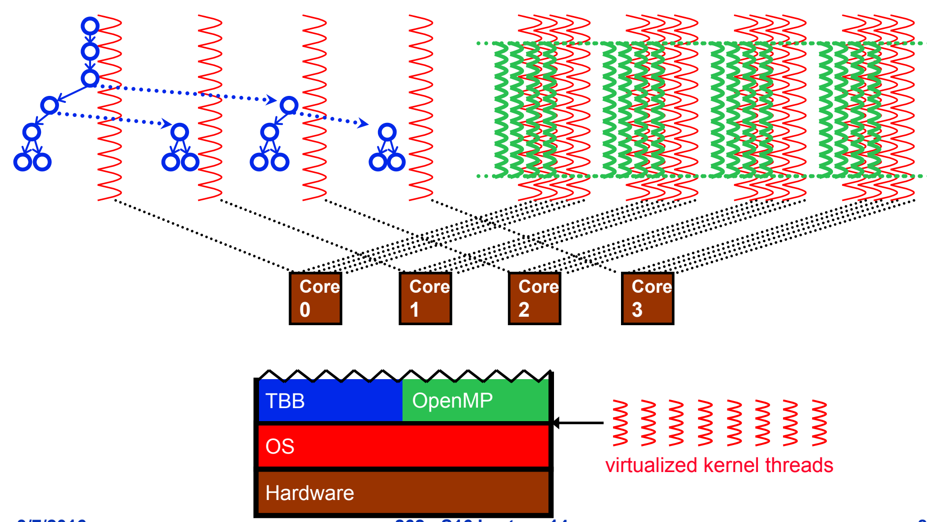
- **Intel's Threading Building Blocks (TBB)**
  - Library that allows programmers to express parallelism using a higher-level, task-based, abstraction
  - Uses work-stealing internally (i.e. Cilk)
  - Open-source
- **Intel's Math Kernel Library (MKL)**
  - Uses OpenMP for parallelism
- **OpenMP**
  - Allows programmers to express parallelism in the SPMD-style using a combination of compiler directives and a runtime library
  - Creates SPMD teams internally (i.e. UPC)
  - Open-source implementation of OpenMP from GNU (libgomp)

Suboptimal Performance

Performance of SPQR on 16-core AMD Opteron System

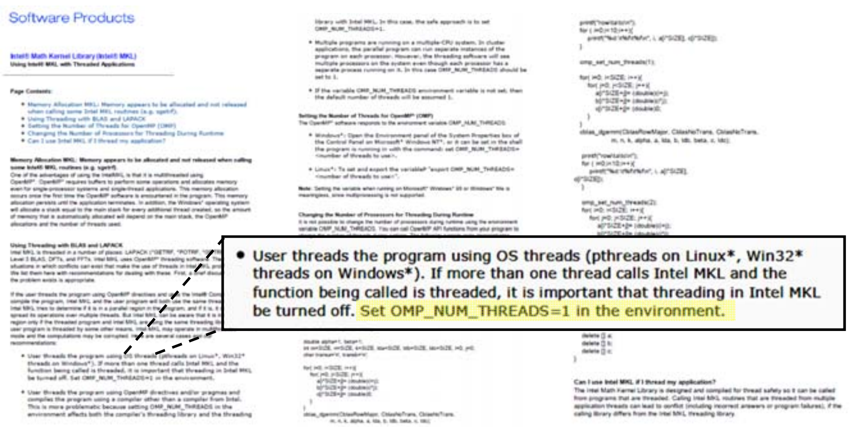


Out-of-the-Box Configurations



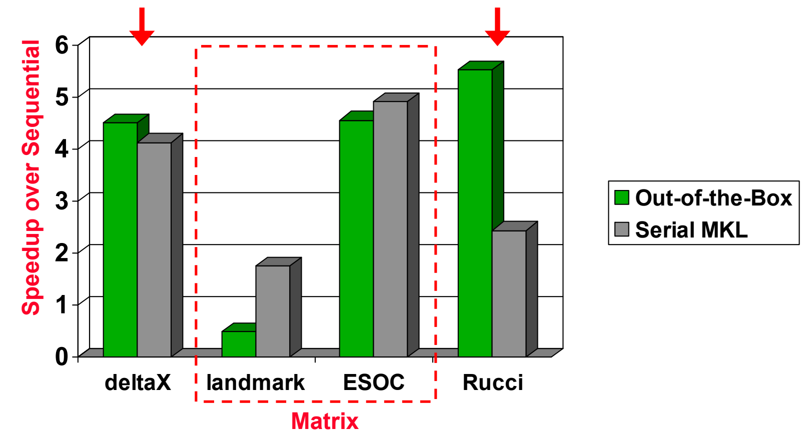
Providing Performance Isolation

Using Intel MKL with Threaded Applications
http://www.intel.com/support/performancetools/libraries/mkl/sb/CS-017177.htm

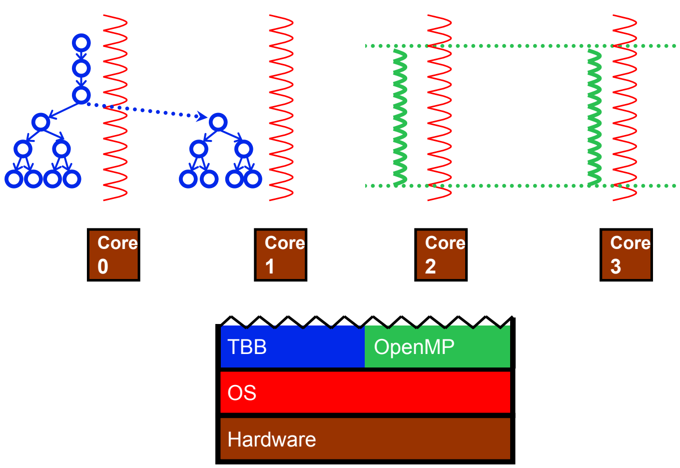


"Tuning" the Code

Performance of SPQR on 16-core AMD Opteron System



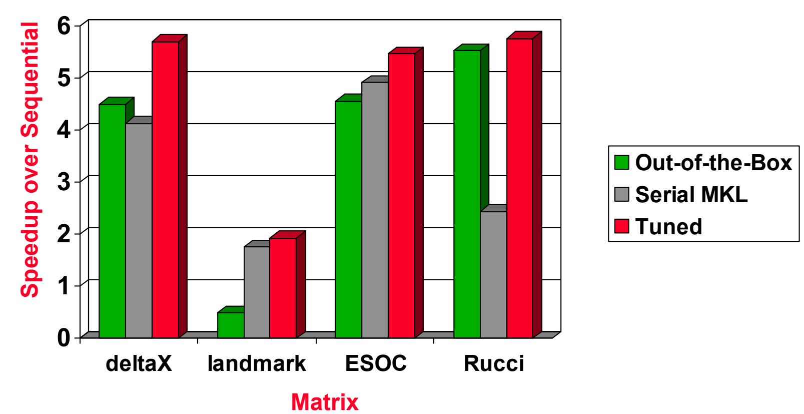
Partition Resources



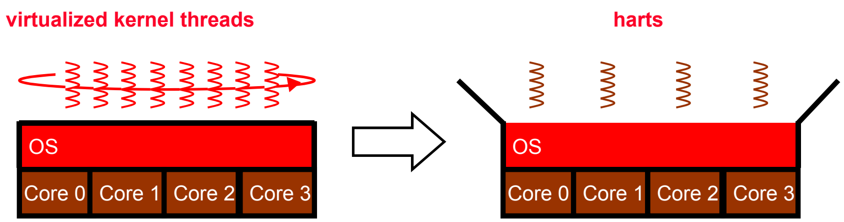
Tim Davis' "tuned" SPQR by manually partitioning the resources.

"Tuning" the Code (continued)

Performance of SPQR on 16-core AMD Opteron System

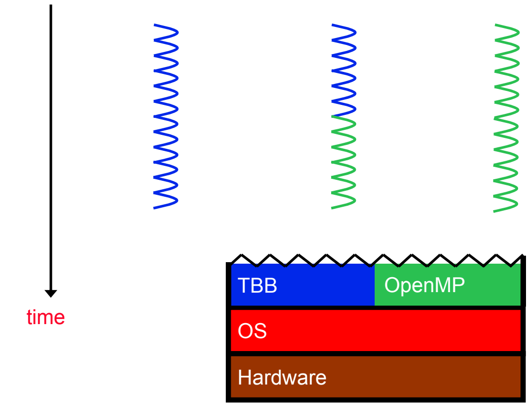


Harts: Hardware Threads

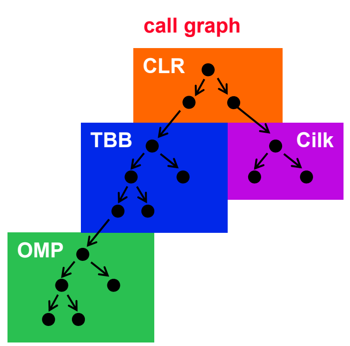


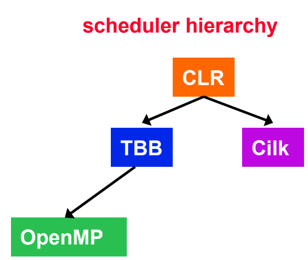
- Expose true hardware resources
  - **Applications requests harts from OS**
  - **Application "schedules" the harts itself (two-level scheduling)**
  - **Can both space-multiplex and time-multiplex harts … but never time-multiplex harts of the same application**

Sharing Harts (Dynamically)



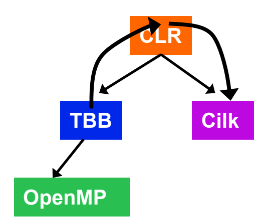
How to Share Harts?



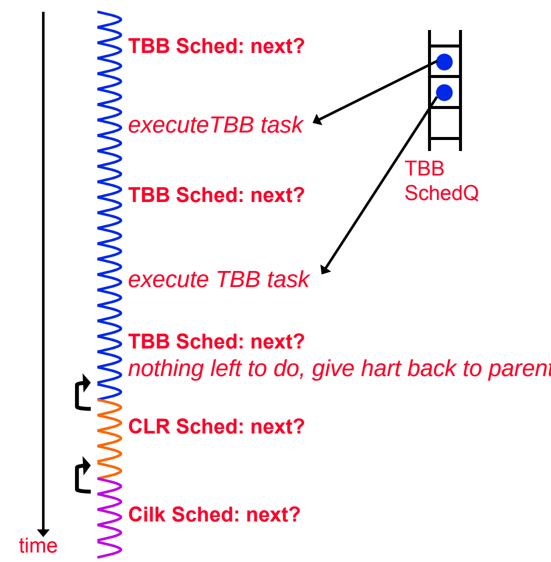


- **Hierarchically:** Caller gives resources to callee to execute
- **Cooperatively:** Callee gives resources back to caller when done

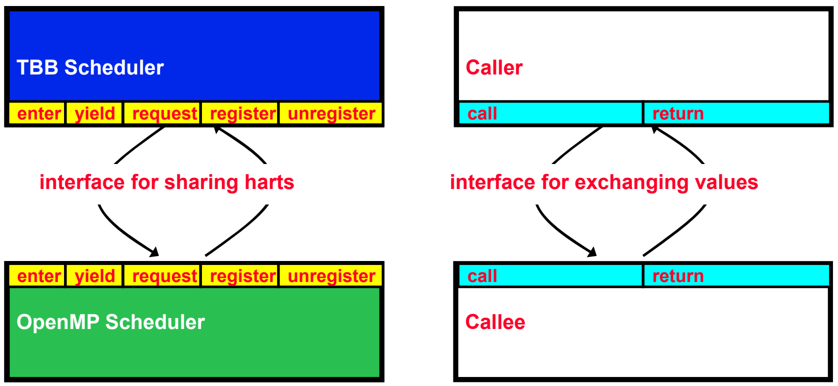
A Day in the Life of a Hart



- **Non-preemptive scheduling.**



Lithe (ABI)

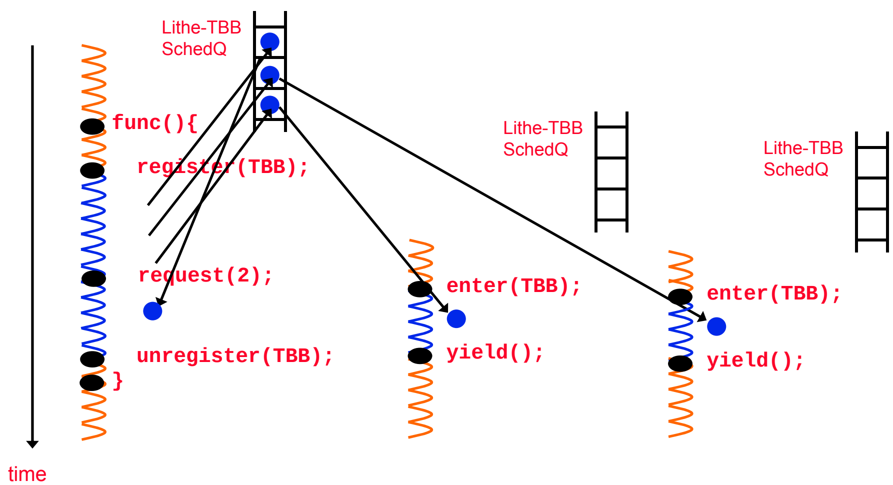


- Analogous to function call ABI for enabling interoperable codes.

A Few Details …

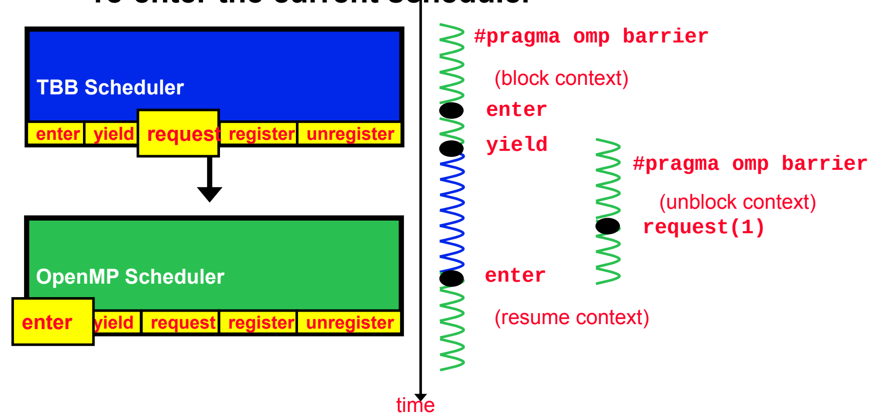
- A hart is only managed by one scheduler at a time
- The Lithe runtime manages the hierarchy of schedulers and the interaction between schedulers
- Lithe ABI only a mechanism to share harts, not policy

Putting It All Together



Synchronization

- Can't block a hart on a synchronization object
- Synchronization objects are implemented by saving the current "context" and having the hart re-enter the current scheduler



Lithe Contexts

- Includes notion of a stack
- Includes context-local storage
- There is a special transition context for each hart that allows it to transition between schedulers easily (i.e. on an enter, yield)

Lithe-compliant Schedulers

- TBB
  - Worker model
  - ~180 lines added, ~5 removed, ~70 modified (~1,500 / ~8,000 total)
- OpenMP
  - Team model
  - ~220 lines added, ~35 removed, ~150 modified (~1,000 / ~6,000 total)

Overheads?

- TBB
  - Example micro-benchmarks that Intel includes with releases

	tree sum	preorder	fibonacci
Lithe-Compliant TBB	54.80	228.20	8.421
TBB	54.80	242.51	8.722

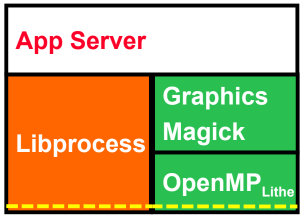
- OpenMP
  - NAS benchmarks (conjugate gradient, LU solver, and multigrid)

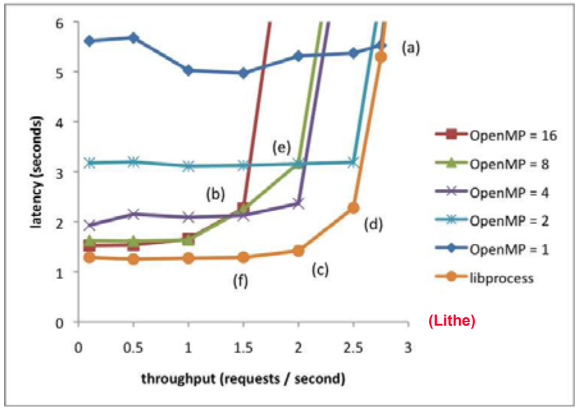
	conjugate gradient (cg)	LU solver (lu)	multigrid (mg)
Lithe-Compliant GNU OpenMP	57.06	122.15	9.23
GNU OpenMP	57.00	123.68	9.54

Flickr Application Server

- GraphicsMagick parallelized using OpenMP
- Server component parallelized using threads (or libprocess processes)
- Spectrum of possible implementations:
  - Process one image upload at a time, pass all resources to OpenMP (via GraphicsMagick)
    - + Easy implementation
    - - Can't overlap communication with computation, some network links are slow, images are different sizes, diminishing returns on resize operations
  - Process as many images as possible at a time, run GraphicsMagick sequentially
    - + Also easy implementation
    - - Really bad latency when low-load on server, 32 core machine underwhelmed
  - All points in between …
    - + Account for changing load, different image sizes, different link bandwidth/latency
    - - Hard to program

Flickr-Like App Server





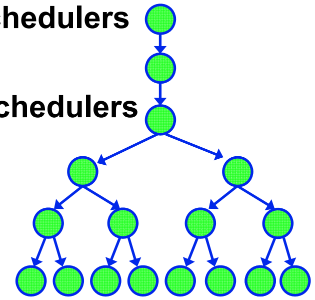
Tradeoff between throughput saturation point and latency.

Case Study: Sparse QR Factorization

- Different matrix sizes

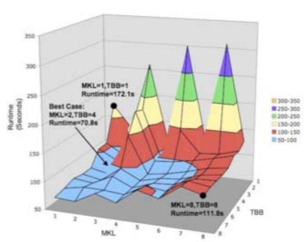
	landmark	deltaX	ESOC	Rucci1
Size	71,952 x 2,704	68,600 x 21,961	327,062 x 37,830	1,977,885 x 109,900
Nonzeros	1,146,868	247,424	6,019,939	7,791,168
Domain	surveying	computer graphics	orbit estimates	ill-conditioned least-square

- deltaX creates ~30,000 OpenMP schedulers
- …
- Rucci creates ~180,000 OpenMP schedulers
- Platform: Dual-socket 2.66 GHz Intel Xeon (Clovertown) with 4 cores per socket (8 total cores)



Case Study: Sparse QR Factorization

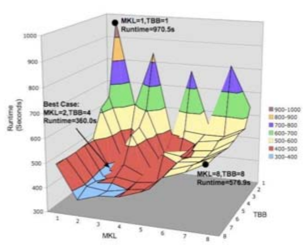
ESOC



Tuned:	70.8
Out-of-the-box:	111.8
Sequential:	172.1

Lithe: 66.7

Rucci

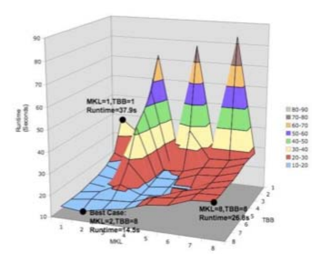


Tuned:	360.0
Out-of-the-box:	576.9
Sequential:	970.5

Lithe: 354.7

Case Study: Sparse QR Factorization

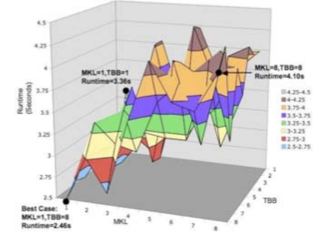
deltaX



Tuned:	14.5
Out-of-the-box:	26.8
Sequential:	37.9

Lithe: 13.6

landmark



Tuned:	2.5
Out-of-the-box:	4.1
Sequential:	3.4

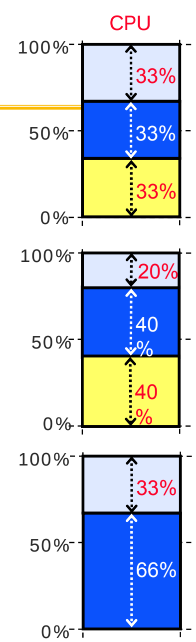
Lithe: 2.3

Is this a good paper?

- What were the authors' goals?
- What about the evaluation/metrics?
- Did they convince you that this was a good system/approach?
- Were there any red-flags?
- What mistakes did they make?
- Does the system/approach meet the "Test of Time" challenge?
- How would you review this paper today?

Break

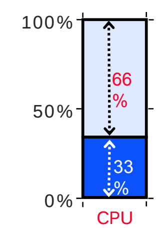
What is Fair Sharing?

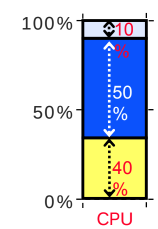


- n users want to share a resource (e.g., CPU)
  - Solution:
    Allocate each 1/n of the shared resource
- *Generalized by max-min fairness*
  - Handles if a user wants less than its fair share
  - E.g. user 1 wants no more than 20%
- Generalized by *weighted max-min fairness*
  - Give weights to users according to importance
  - User 1 gets weight 1, user 2 weight 2

Why is Fair Sharing Useful?

- *Weighted Fair Sharing* / *Proportional Shares*
  - User 1 gets weight 2, user 2 weight 1
- *Priorities*
  - Give user 1 weight 1000, user 2 weight 1
- *Revervations*
  - Ensure user 1 gets 10% of a resource
  - Give user 1 weight 10, sum weights ≤ 100
- *Isolation Policy*
  - Users cannot affect others beyond their fair share





Properties of Max-Min Fairness

- Share guarantee
  - Each user can get at least 1/n of the resource
  - But will get less if her demand is less
- Strategy-proof
  - Users are not better off by asking for more than they need
  - Users have no reason to lie
- Max-min fairness is the only "reasonable" mechanism with these two properties

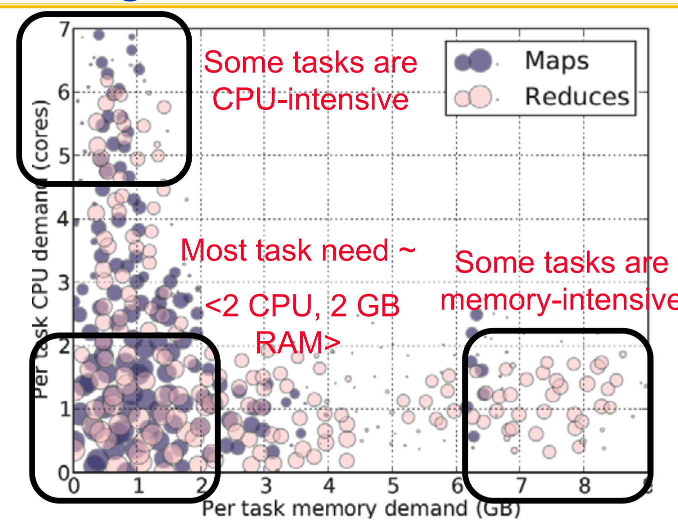
Why Care about Fairness?

- Desirable properties of max-min fairness
  - *Isolation policy:* A user gets her fair share irrespective of the demands of other users
  - *Flexibility separates mechanism from policy:* Proportional sharing, priority, reservation,...
- *Many schedulers* use max-min fairness
  - Datacenters: Hadoop's fair sched, capacity, Quincy
  - OS: rr, prop sharing, lottery, linux cfs, ...
  - Networking: wfq, wf2q, sfq, drr, csfq, ...

When is Max-Min Fairness not Enough?

- Need to schedule *multiple, heterogeneous* resources
  - Example: Task scheduling in datacenters
    - » Tasks consume more than just CPU – CPU, memory, disk, and I/O
- What are today's datacenter task demands?

Heterogeneous Resource Demands

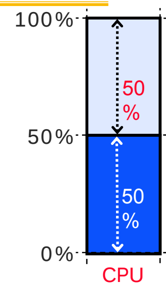


2000-node Hadoop Cluster at Facebook (Oct 2010)

Problem

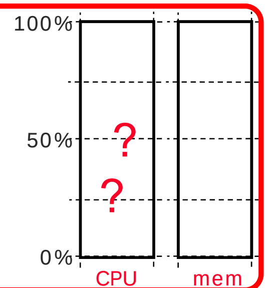
Single resource example

– 1 resource: CPU
– User 1 wants <1 CPU> per task
– User 2 wants <3 CPU> per task



Multi-resource example

– 2 resources: CPUs & memory
– User 1 wants <1 CPU, 4 GB> per task
– User 2 wants <3 CPU, 1 GB> per task
– *What is a fair allocation?*



Problem definition

How to fairly share multiple resources when users have heterogeneous demands on them?

Model

- Users have *tasks* according to a *demand vector*
  - e.g. <2, 3, 1> user's tasks need 2 R_1, 3 R_2, 1 R_3
  - Not needed in practice, can simply measure actual consumption
- Resources given in multiples of demand vectors
- Assume divisible resources

What is Fair?

- **Goal: define a fair allocation of multiple cluster resources between multiple users**
- **Example: suppose we have:**
  - **30 CPUs and 30 GB RAM**
  - **Two users with equal shares**
  - **User 1 needs <1 CPU, 1 GB RAM> per task**
  - **User 2 needs <1 CPU, 3 GB RAM> per task**
- **What is a fair allocation?**

First Try: Asset Fairness

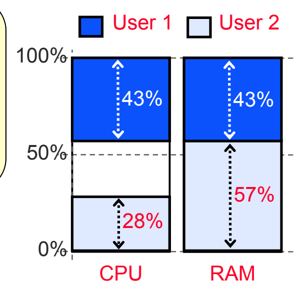
- *Asset Fairness*
  – Equalize each user's *sum of resource shares*

Problem

User 1 has < 50% of both CPUs and RAM

Better off in a separate cluster with 50% of the resources

- Asset fairness yields
  – U_1: 15 tasks: 30 CPUs, 30 GB (∑=60)
  – U_2: 20 tasks: 20 CPUs, 40 GB (∑=60)



Lessons from Asset Fairness

"You shouldn't do worse than if you ran a smaller, private cluster equal in size to your fair share"

Thus, given N users, each user should get ≥ 1/N of her dominating resource (i.e., the resource that she consumes most of)

Desirable Fair Sharing Properties

- Many desirable properties
  – Share Guarantee
  – Strategy proofness
  – Envy-freeness
  – Pareto efficiency
  – Single-resource fairness
  – Bottleneck fairness
  – Population monotonicity
  – Resource monotonicity

DRF focuses on these properties

Cheating the Scheduler

- Some users will *game* the system to get more resources

- Real-life examples
  – A cloud provider had quotas on map and reduce slots
  Some users found out that the map-quota was low
    » Users implemented maps in the reduce slots!

  – A search company provided dedicated machines to users that could ensure certain level of utilization (e.g. 80%)
    » Users used busy-loops to inflate utilization

Two Important Properties

- Strategy-proofness
  - A user should not be able to increase her allocation by lying about her demand vector
  - Intuition:
    - » Users are incentivized to make truthful resource requirements
- Envy-freeness
  - No user would ever strictly prefer another user's lot in an allocation
  - Intuition:
    - » Don't want to trade places with any other user

Challenge

- A fair sharing policy that provides
  - Strategy-proofness
  - Share guarantee
- Max-min fairness for a single resource had these properties
  - Generalize max-min fairness to multiple resources

Dominant Resource Fairness

- A user's *dominant resource* is the resource she has the biggest share of
  - Example:

    Total resources: <10 CPU, 4 GB>

    User 1's allocation: <2 CPU, 1 GB>

    Dominant resource is memory as 1/4 > 2/10 (1/5)
- A user's *dominant share* is the fraction of the dominant resource she is allocated
  - User 1's dominant share is 25% (1/4)

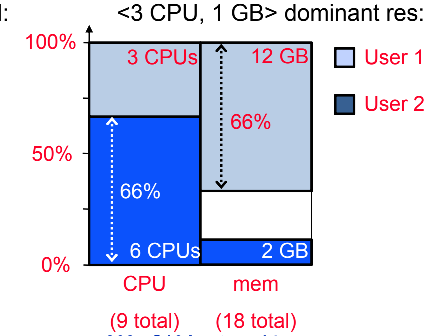
Dominant Resource Fairness (2)

- *Apply max-min fairness to dominant shares*
- Equalize the dominant share of the users
  - Example:

    Total resources: <9 CPU, 18 GB>

    User 1 demand: <1 CPU, 4 GB> dominant res: mem

    User 2 demand: <3 CPU, 1 GB> dominant res: CPU



DRF is Fair

- DRF is strategy-proof
- DRF satisfies the share guarantee
- DRF allocations are envy-free

See DRF paper for proofs

Online DRF Scheduler

Whenever there are available resources and tasks to run:

Schedule a task to the user with smallest dominant share

- O(log *n*) time per decision using binary heaps
- Need to determine demand vectors

Alternative: Use an Economic Model

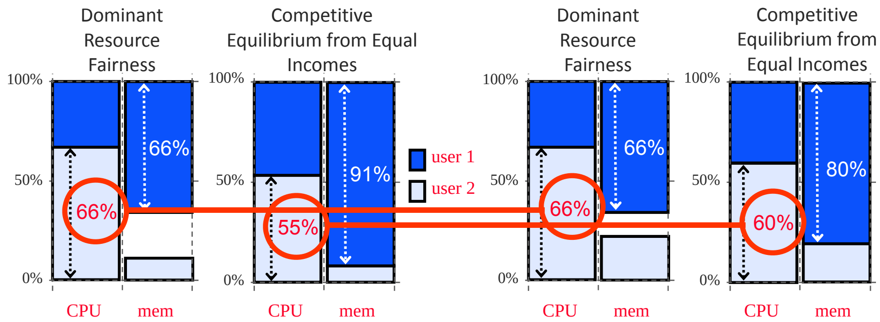
- Approach
  - Set prices for each good
  - Let users buy what they want
- How do we determine the right prices for different goods?
- Let the market determine the prices
- *Competitive Equilibrium from Equal Incomes (CEEI)*
  - Give each user 1/n of every resource
  - Let users trade in a perfectly competitive market
- Not strategy-proof!

Determining Demand Vectors

- They can be *measured*
  - Look at actual resource consumption of a user
- They can be *provided* the by user
  - What is done today
- In both cases, strategy-proofness incentivizes user to consume resources wisely

DRF vs CEEI

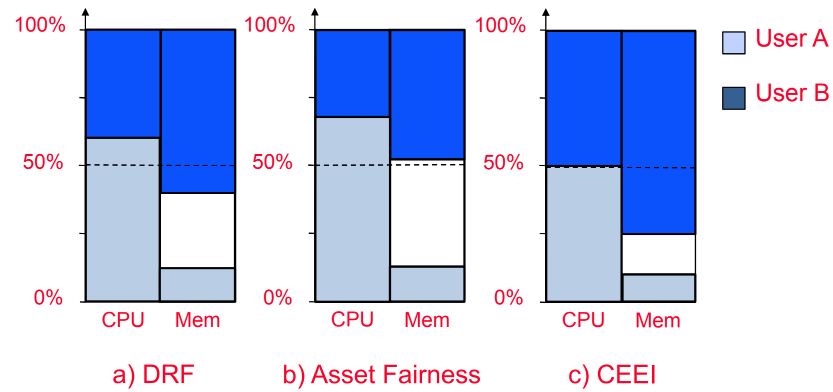
- User 1: <1 CPU, 4 GB> User 2: <3 CPU, 1 GB>
  - DRF more fair, CEEI better utilization



- User 1: <1 CPU, 4 GB> User 2: <3 CPU, 2 GB>
  - User 2 increased her share of both CPU and memory

Example of DRF vs Asset vs CEEI

- Resources <1000 CPUs, 1000 GB>
- 2 users A: <2 CPU, 3 GB> and B: <5 CPU, 1 GB>



Desirable Fairness Properties (1)

- Recall *max/min fairness* from networking
  - Maximize the bandwidth of the minimum flow [Bert92]

- *Progressive filling (PF) algorithm*
  1. Allocate ε to every flow until some link saturated
  2. Freeze allocation of all flows on saturated link and goto 1

Desirable Fairness Properties (2)

- *P1. Pareto Efficiency*
  » It should not be possible to allocate more resources to any user without hurting others

- *P2. Single-resource fairness*
  » If there is only one resource, it should be allocated according to max/min fairness

- *P3. Bottleneck fairness*
  » If all users want most of one resource(s), that resource should be shared according to max/min fairness

Desirable Fairness Properties (3)

- Assume *positive demands* ($D_{ij} > 0$ for all i and j)
- DRF will allocate same dominant share to all users
  - As soon as PF saturates a resource, allocation frozen

Desirable Fairness Properties (4)

- *P4. Population Monotonicity*
  - If a user leaves and relinquishes her resources, no other user's allocation should get hurt
  - Can happen each time a job finishes
- CEEI violates population monotonicity
- DRF satisfies population monotonicity
  - Assuming positive demands
  - Intuitively DRF gives the same dominant share to all users, so all users would be hurt contradicting Pareto efficiency

Properties of Policies

Property	Asset	CEEI	DRF
Share guarantee		✓	✓
Strategy-proofness	✓		✓
Pareto efficiency	✓	✓	✓
Envy-freeness	✓	✓	✓
Single resource fairness	✓	✓	✓
Bottleneck res. fairness		✓	✓
Population monotonicity	✓		✓
Resource monotonicity			

Evaluation Methodology

- Micro-experiments on EC2
  - Evaluate DRF's dynamic behavior when demands change
  - Compare DRF with current Hadoop scheduler
- Macro-benchmark through simulations
  - Simulate Facebook trace with DRF and current Hadoop scheduler

DRF Inside Mesos on EC2

Dominant resource is memory

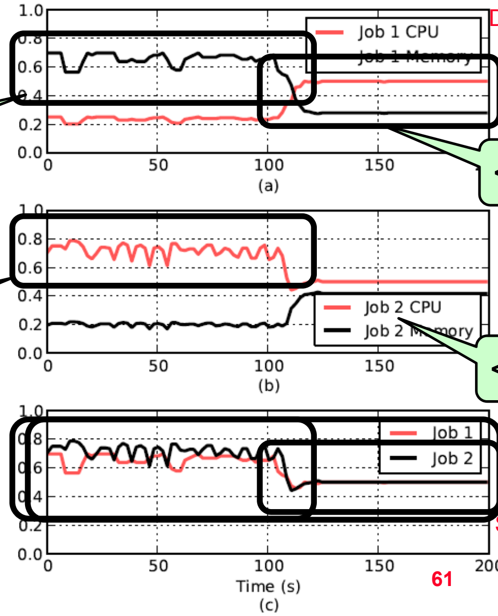
<1 CPU, 10 GB>

Dominant resource is CPU

<1 CPU, 1 GB>

Dominant shares are equalized

Share guarantee: ~70% dominant share



Dominant resource is CPU

User 1's Shares

<2 CPU, 4 GB>

User 2's Shares

<1 CPU, 3 GB>

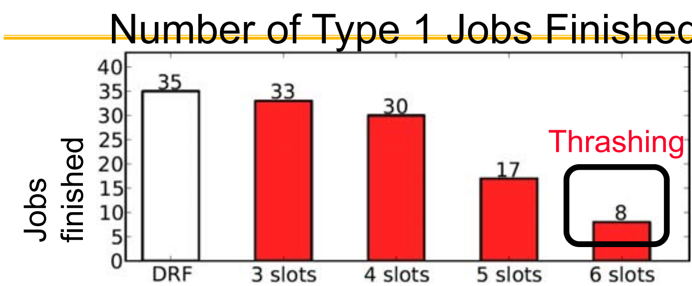
Dominant Shares

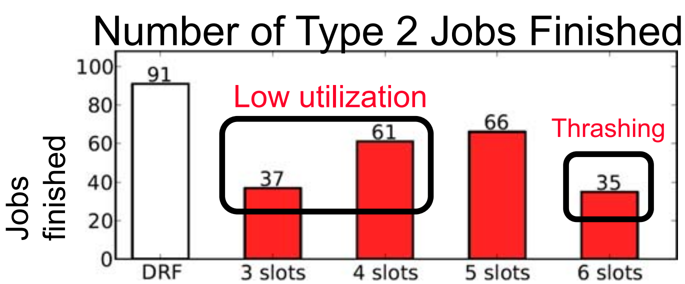
Share guarantee: ~50% dominant share

Fairness in Today's Datacenters

- Hadoop Fair Scheduler/capacity/Quincy
  - Each machine consists of k *slots* (e.g. k=14)
  - Run at most one task per slot
  - Give jobs "equal" number of slots, i.e., apply max-min fairness to slot-count
- This is what DRF paper compares against

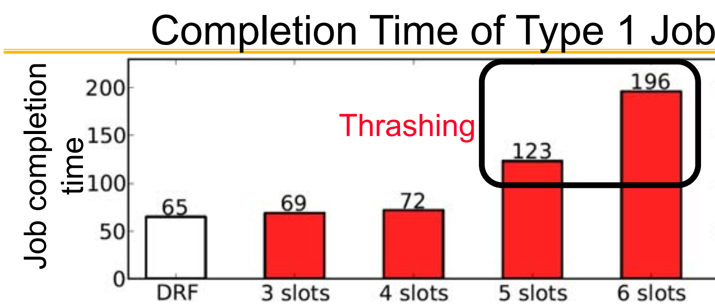
Experiment: DRF vs Slots

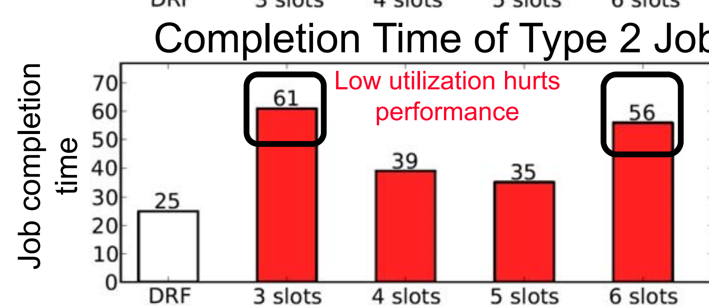




Type 1 jobs <2 CPU, 2 GB> Type 2 jobs <1 CPU, 0.5GB>

Experiment: DRF vs Slots



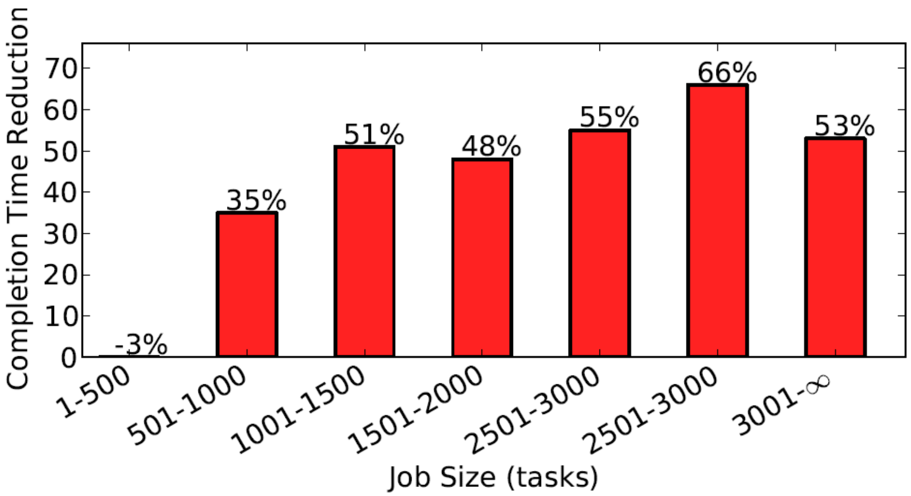


Thrashing

Type 1 job <2 CPU, 2 GB> Type 2 job <1 CPU, 0.5GB>

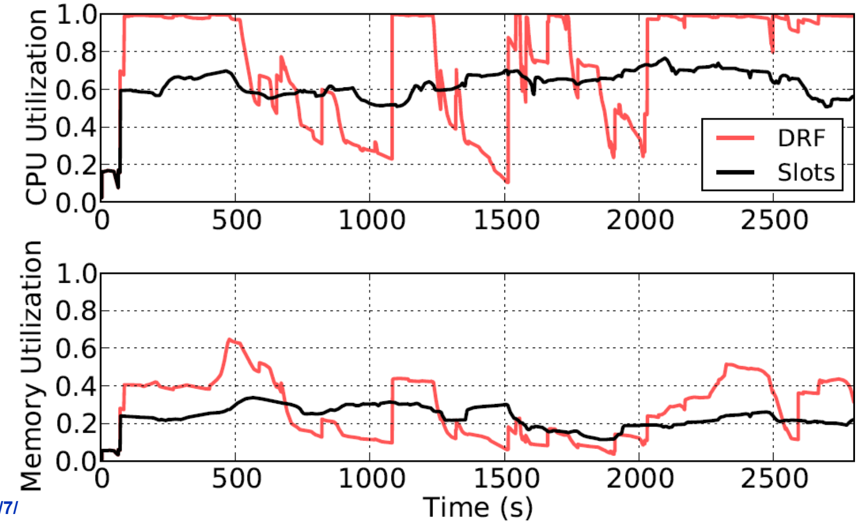
Reduction in Job Completion Time DRF vs Slots

- Simulation of 1-week Facebook traces



Utilization of DRF vs Slots

- Simulation of Facebook workload



Summary

- DRF provides *multiple-resource fairness* in the presence of *heterogeneous demand*
  - First generalization of max-min fairness to multiple-resources
- DRF's properties
  - Share guarantee, at least 1/n of one resource
  - Strategy-proofness, lying can only hurt you
  - Performs better than current approaches

Is this a good paper?

- What were the authors' goals?
- What about the evaluation/metrics?
- Did they convince you that this was a good system/approach?
- Were there any red-flags?
- What mistakes did they make?
- Does the system/approach meet the "Test of Time" challenge?
- How would you review this paper today?