

# **Workload-Driven Architectural Evaluation**

# Evaluation in Uniprocessors

Decisions made only after quantitative evaluation

For existing systems: comparison and procurement evaluation

For future systems: careful extrapolation from known quantities

Wide base of programs leads to standard benchmarks

- Measured on wide range of machines and successive generations

Measurements and technology assessment lead to proposed features

Then simulation

- Simulator developed that can run with and without a feature
- Benchmarks run through the simulator to obtain results
- Together with cost and complexity, decisions made

# Difficult Enough for Uniprocessors

Workloads need to be renewed and reconsidered

Input data sets affect key interactions

- Changes from SPEC92 to SPEC95

Accurate simulators costly to develop and verify

Simulation is time-consuming

But the effort pays off: Good evaluation leads to good design

Quantitative evaluation increasingly important for multiprocessors

- Maturity of architecture, and greater continuity among generations
- It's a grounded, engineering discipline now

Good evaluation is critical, and we must learn to do it right

# More Difficult for Multiprocessors

What is a representative workload?

Software model has not stabilized

Many architectural and application degrees of freedom

- Huge design space: no. of processors, other architectural, application
- Impact of these parameters and their interactions can be huge
- High cost of communication

What are the appropriate metrics?

Simulation is expensive

- Realistic configurations and sensitivity analysis difficult
- Larger design space, but more difficult to cover

Understanding of parallel programs as workloads is critical

- Particularly interaction of application and architectural parameters

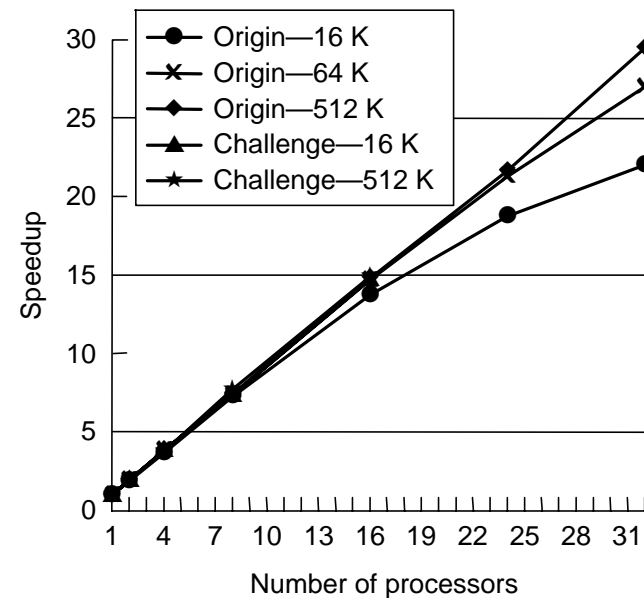
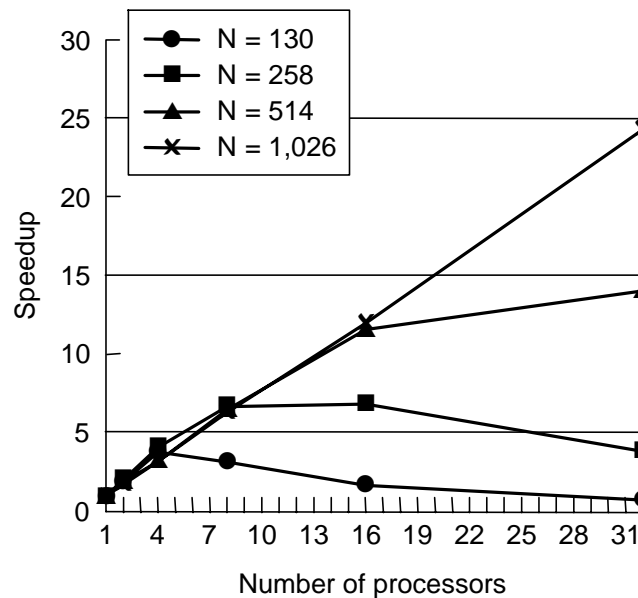
# A Lot Depends on Sizes

Application parameters and no. of procs affect inherent properties

- Load balance, communication, extra work, temporal and spatial locality

Interactions with organization parameters of extended memory hierarchy affect artifactual communication and performance

Effects often dramatic, sometimes small: application-dependent



Understanding size interactions and scaling relationships is key

# Outline

Performance and scaling (of workload and architecture)

- Techniques
- Implications for behavioral characteristics and performance metrics

Evaluating a real machine

- Choosing workloads
- Choosing workload parameters
- Choosing metrics and presenting results

Evaluating an architectural idea/tradeoff through simulation

Public-domain workload suites

Characteristics of our applications

# Measuring Performance

Absolute performance

- Most important to end user

Performance improvement due to parallelism

- $Speedup(p) = Performance(p) / Performance(1)$ , always

Both should be measured

$Performance = Work / Time$ , always

Work is determined by input configuration of the problem

If work is fixed, can measure performance as  $1/Time$

- Or retain explicit work measure (e.g. transactions/sec, bonds/sec)
- Still w.r.t particular configuration, and still what's measured is time

$$\bullet \text{ Speedup}(p) = \frac{\text{Time}(1)}{\text{Time}(p)} \quad \text{or} \quad \frac{\text{Operations Per Second } (p)}{\text{Operations Per Second } (1)}$$

# Scaling: Why Worry?

Fixed problem size is limited

Too small a problem:

- May be appropriate for small machine
- Parallelism overheads begin to dominate benefits for larger machines
  - Load imbalance
  - Communication to computation ratio
- May even achieve slowdowns
- Doesn't reflect real usage, and inappropriate for large machines
  - Can exaggerate benefits of architectural improvements, especially when measured as percentage improvement in performance

Too large a problem

- Difficult to measure improvement (next)

# Too Large a Problem

Suppose problem realistically large for big machine

May not “fit” in small machine

- Can't run
- Thrashing to disk
- Working set doesn't fit in cache

Fits at some  $p$ , leading to *superlinear speedup*

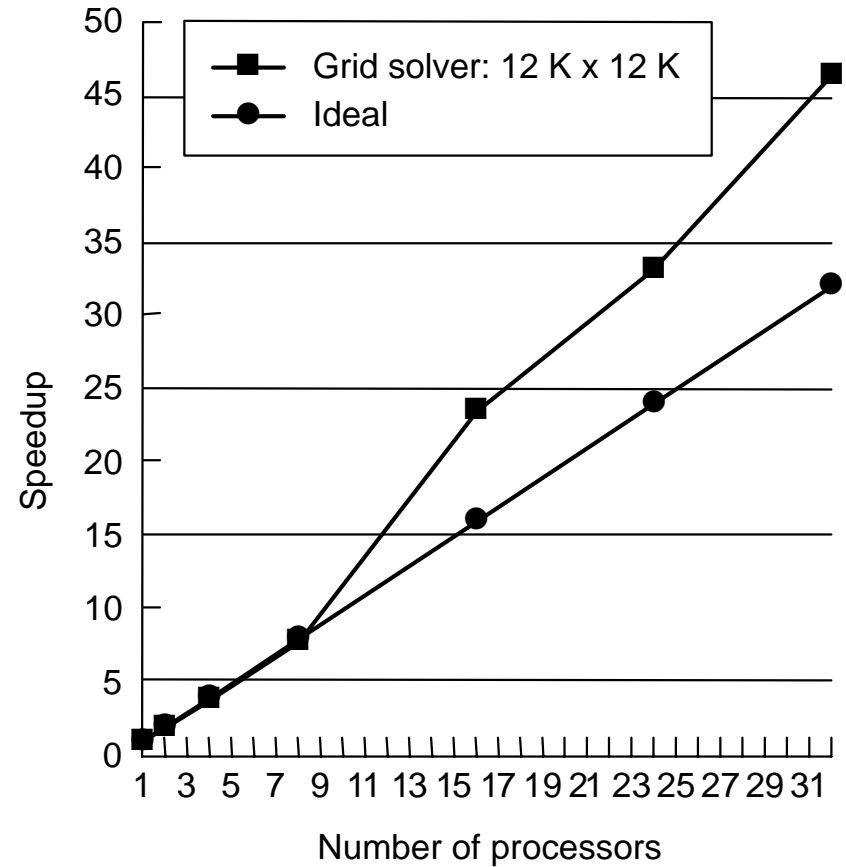
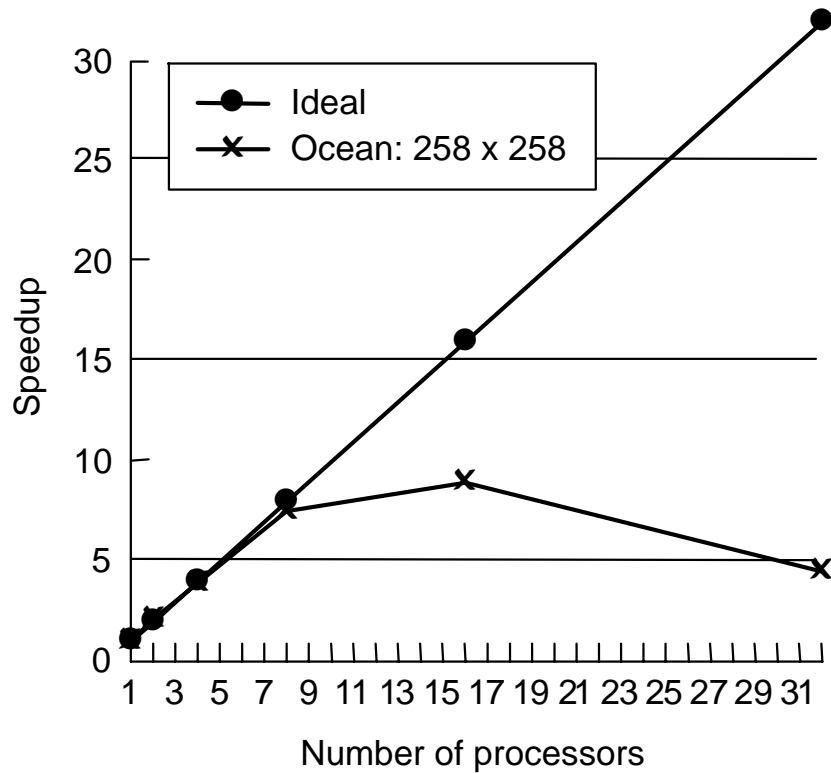
Real effect, but doesn't help evaluate effectiveness

Finally, users want to scale problems as machines grow

- Can help avoid these problems

# Demonstrating Scaling Problems

Small Ocean and big equation solver problems on SGI Origin2000



# Questions in Scaling

Under what constraints to scale the application?

- What are the appropriate metrics for performance improvement?
  - work is not fixed any more, so time not enough

How should the application be scaled?

Definitions:

*Scaling a machine*: Can scale power in many ways

- Assume adding identical nodes, each bringing memory

*Problem size*: Vector of input parameters, e.g.  $N = (n, q, \Delta t)$

- Determines work done
- Distinct from *data set size* and *memory usage*
- Start by assuming it's only one parameter  $n$ , for simplicity

# Under What Constraints to Scale?

Two types of constraints:

- User-oriented, e.g. particles, rows, transactions, I/Os per processor
- Resource-oriented, e.g. memory, time

Which is more appropriate depends on application domain

- User-oriented easier for user to think about and change
- Resource-oriented more general, and often more real

Resource-oriented scaling models:

- *Problem constrained* (PC)
- *Memory constrained* (MC)
- *Time constrained* (TC)

(TPC: transactions, users, terminals scale with “computing power”)

Growth under MC and TC may be hard to predict

# Problem Constrained Scaling

User wants to solve same problem, only faster

- Video compression
- Computer graphics
- VLSI routing

But limited when evaluating larger machines

$$Speedup_{PC}(p) = \frac{Time(1)}{Time(p)}$$

# Time Constrained Scaling

Execution time is kept fixed as system scales

- User has fixed time to use machine or wait for result

Performance = Work/Time as usual, and time is fixed, so

$$SpeedupTC(p) = \frac{Work(p)}{Work(1)}$$

How to measure work?

- Execution time on a single processor? (thrashing problems)
- Should be easy to measure, ideally analytical and intuitive
- Should scale linearly with sequential complexity
  - Or ideal speedup will not be linear in  $p$  (e.g. no. of rows in matrix program)
- If cannot find intuitive application measure, as often true, measure *execution time with ideal memory system on a uniprocessor* (e.g. pixie)

# Memory Constrained Scaling

Scale so memory usage per processor stays fixed

Scaled Speedup:  $\text{Time}(1) / \text{Time}(p)$  for scaled up problem

- Hard to measure  $\text{Time}(1)$ , and inappropriate

$$\text{Speedup}_{MC}(p) = \frac{\text{Work}(p)}{\text{Time}(p)} \times \frac{\text{Time}(1)}{\text{Work}(1)} = \frac{\text{Increase in Work}}{\text{Increase in Time}}$$

Can lead to large increases in execution time

- If work grows faster than linearly in memory usage
- e.g. matrix factorization
  - 10,000-by 10,000 matrix takes 800MB and 1 hour on uniprocessor
  - With 1,000 processors, can run 320K-by-320K matrix, but ideal parallel time grows to 32 hours!
  - With 10,000 processors, 100 hours ...

Time constrained seems to be most generally viable model

# Impact of Scaling Models: Grid Solver

## MC Scaling:

- Grid size =  $n^3/p$ -by- $n^3/p$
- Iterations to converge =  $n^3/p$
- Work =  $O(n^3/p)^3$
- Ideal parallel execution time =  $O\left(\frac{(n^3/p)^3}{p}\right) = n^9/p^4$
- Grows by  $n^3/p$
- 1 hr on uniprocessor means 32 hr on 1024 processors

## TC scaling:

- If scaled grid size is  $k$ -by- $k$ , then  $k^3/p = n^3$ , so  $k = n^3/p$ .
- Memory needed per processor =  $k^2/p = n^6/p^2$
- Diminishes as cube root of number of processors

# Impact on Solver Execution Characteristics

Concurrency: PC: fixed; MC: grows as  $p$ ; TC: grows as  $p^{0.67}$

Comm to comp: PC: grows as  $\sqrt{p}$ ; MC: fixed; TC: grows as  $\sqrt[6]{p}$

Working Set: PC: shrinks as  $p$ ; MC: fixed; TC: shrinks as  $\sqrt[3]{p}$

Spatial locality?

Message size in message passing?

- Expect speedups to be best under MC and worst under PC
- Should evaluate under all three models, unless some are unrealistic

# Scaling Workload Parameters: Barnes-Hut

Different parameters govern different sources of error:

- Number of bodies  $(n)$
- Time-step resolution  $(\Delta t)$
- Force-calculation accuracy  $(\theta)$

Scaling rule:

All components of simulation error should scale at same rate

Result: If  $n$  scales by a factor of  $s$

- $\Delta t$  and  $\theta$  must both scale by a factor of  $\frac{1}{s^2}$

# Effects of Scaling Rule

If number of processors ( $p$ ) is scaled by a factor of  $k$

- Under Time Constrained scaling:
  - increase in number of particles is less than " $k$ "
- Under Memory Constrained scaling:
  - elapsed time becomes unacceptably large

Time Constrained is most realistic for this application

Effect on execution characteristics

- Each parameter has its own effect on c-to-c ratio, working sets etc.
- Scaling them inappropriately can lead to incorrect results
- e.g. working set in Barnes-Hut is  $\frac{1}{\theta^2} \log n$ 
  - With proper scaling, grows much more quickly with  $\theta$  than with  $n$ , so scaling only  $n$  would give misleading results
  - Unlike solver, working set independent. of  $p$ ; increases under TC scaling

# Performance and Scaling Summary

Performance improvement due to parallelism measured by speedup

Scaling models are fundamental to proper evaluation

Speedup metrics take different forms for different scaling models

Scaling constraints affect growth rates of key execution properties

- Time constrained scaling is a realistic method for many applications

Should scale workload parameters appropriately with one another too

- Scaling only data set size can yield misleading results

Proper scaling requires understanding the workload

# Evaluating a Real Machine

Performance Isolation using Microbenchmarks

Choosing Workloads

Evaluating a Fixed-size Machine

Varying Machine Size

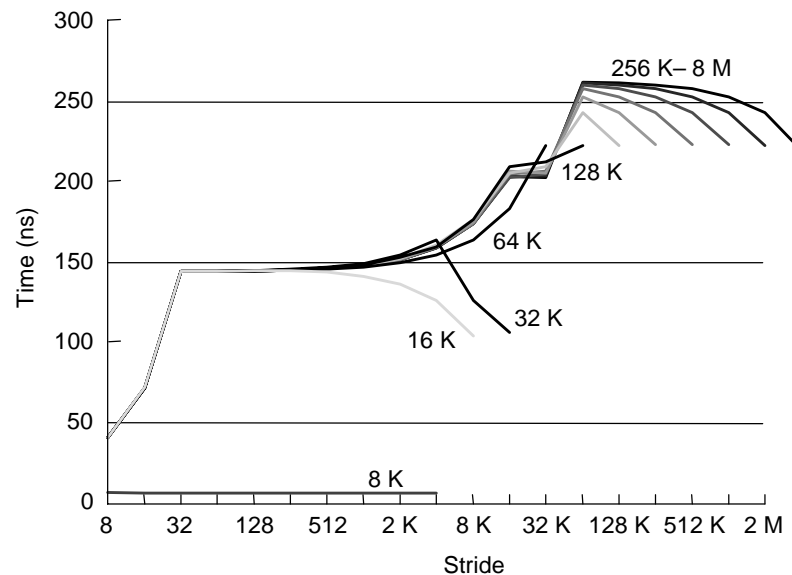
Metrics

- All these issues, plus more, relevant to evaluating a tradeoff via simulation

# Performance Isolation: Microbenchmarks

Microbenchmarks: Small, specially written programs to isolate performance characteristics

- Processing
- Local memory
- Input/output
- Communication and remote access (read/write, send/receive)
- Synchronization (locks, barriers)
- Contention



# Evaluation using Realistic Workloads

Must navigate three major axes:

- Workloads
- Problem Sizes
- No. of processors (one measure of machine size)  
(other machine parameters are fixed)

Focus first on fixed number of processors

# Types of Workloads

- *Kernels*: matrix factorization, FFT, depth-first tree search
- *Complete Applications*: ocean simulation, crew scheduling, database
- *Multiprogrammed Workloads*

**Multiprog.** ↔ **Appls** ↔ **Kernels** ↔ **Microbench.**

Realistic  
Complex  
Higher level interactions  
Are what really matters

Easier to understand  
Controlled  
Repeatable  
Basic machine characteristics

Each has its place:

*Use kernels and microbenchmarks to gain understanding, but applications to evaluate effectiveness and performance*

# Desirable Properties of Workloads

Representativeness of application domains

Coverage of behavioral properties

Adequate concurrency

# Representativeness

Should adequately represent domains of interest, e.g.:

- *Scientific*: Physics, Chemistry, Biology, Weather ...
- *Engineering*: CAD, Circuit Analysis ...
- *Graphics*: Rendering, radiosity ...
- *Information management*: Databases, transaction processing, decision support ...
- *Optimization*
- *Artificial Intelligence*: Robotics, expert systems ...
- *Multiprogrammed general-purpose workloads*
- *System software*: e.g. the operating system

# Coverage: Stressing Features

Easy to mislead with workloads

- Choose those with features for which machine is good, avoid others

Some features of interest:

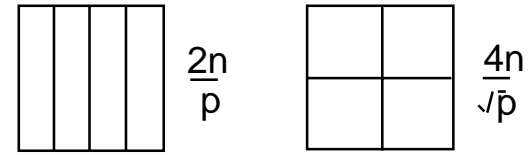
- Compute v. memory v. communication v. I/O bound
- Working set size and spatial locality
- Local memory and communication bandwidth needs
- Importance of communication latency
- Fine-grained or coarse-grained
  - Data access, communication, task size
- Synchronization patterns and granularity
- Contention
- Communication patterns

Choose workloads that cover a range of properties

# Coverage: Levels of Optimization

Many ways in which an application can be suboptimal

- *Algorithmic*, e.g. assignment, blocking



- *Data structuring*, e.g. 2-d or 4-d arrays for SAS grid problem
- *Data layout, distribution and alignment*, even if properly structured
- *Orchestration*
  - contention
  - long versus short messages
  - synchronization frequency and cost, ...
- Also, random problems with “unimportant” data structures

Optimizing applications takes work

- Many practical applications may not be very well optimized

May examine selected different levels to test robustness of system

# Concurrency

Should have enough to utilize the processors

- If load imbalance dominates, may not be much machine can do
- (Still, useful to know what kinds of workloads/configurations don't have enough concurrency)

*Algorithmic speedup*: useful measure of concurrency/imbalance

- Speedup (under scaling model) assuming all memory/communication operations take zero time
- Ignores memory system, measures imbalance and extra work
- Uses PRAM machine model (Parallel Random Access Machine)
  - Unrealistic, but widely used for theoretical algorithm development

At least, should isolate performance limitations due to program characteristics that a machine cannot do much about (concurrency) from those that it can.

# Workload/Benchmark Suites

*Numerical Aerodynamic Simulation (NAS)*

- Originally pencil and paper benchmarks

*SPLASH/SPLASH-2*

- Shared address space parallel programs

*ParkBench*

- Message-passing parallel programs

*ScaLapack*

- Message-passing kernels

*TPC*

- Transaction processing

*SPEC-HPC*

...

# Evaluating a Fixed-size Machine

Now assume workload is fixed too

Many critical characteristics depend on problem size

- Inherent application characteristics
  - concurrency and load balance (generally improve with problem size)
  - communication to computation ratio (generally improve)
  - working sets and spatial locality (generally worsen and improve, resp.)
- Interactions with machine organizational parameters
- Nature of the major bottleneck: comm., imbalance, local access...

Insufficient to use a single problem size

Need to choose problem sizes appropriately

- Understanding of workloads will help

Examine step by step using grid solver

- Assume 64 processors with 1MB cache and 64MB memory each

# Steps in Choosing Problem Sizes

## 1. Appeal to higher powers

- May know that users care only about a few problem sizes
- But not generally applicable

## 2. Determine range of useful sizes

- Below which bad perf. or unrealistic time distribution in phases
- Above which execution time or memory usage too large

## 3. Use understanding of inherent characteristics

- Communication-to-computation ratio, load balance...
- For grid solver, perhaps at least 32-by-32 points per processor
- 40MB/s c-to-c ratio with 200MHz processor
- No need to go below 5MB/s (larger than 256-by-256 subgrid per processor) from this perspective, or 2K-by-2K grid overall

So assume we choose 256-by-256, 1K-by-1K and 2K-by-2K so far

# Steps in Choosing Problem Sizes (contd.)

Variation of characteristics with problem size usually smooth

- So, for inherent comm. and load balance, pick some sizes along range

Interactions of locality with architecture often have thresholds (knees)

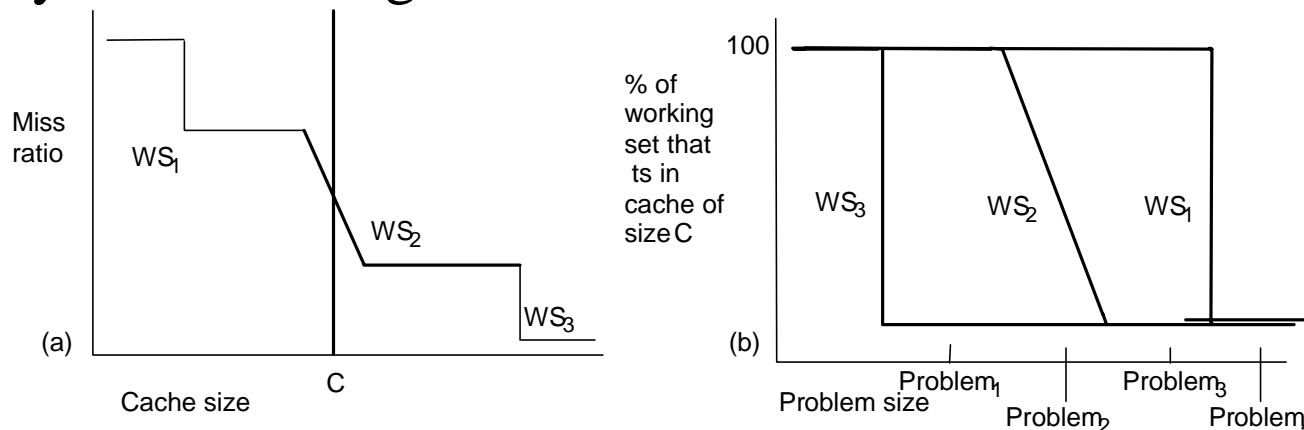
- Greatly affect characteristics like local traffic, artifactual comm.
- May require problem sizes to be added
  - to ensure both sides of a knee are captured
- But also help prune the design space

# Steps in Choosing Problem Sizes (contd.)

## 4. Use temporal locality and working sets

Fitting or not dramatically changes local traffic and artifactual comm.

E.g. Raytrace working sets are nonlocal, Ocean are local



- Choose problem sizes on both sides of a knee if realistic
  - *Critical to understand growth rate of working sets*
- Also try to pick one very large size (exercises TLB misses etc.)
- Solver: first (2 subrows) usually fits, second (full partition) may or not
  - Doesn't for largest (2K) so add 4K-b-4K grid
  - Add 16K as large size, so grid sizes now 256, 1K, 2K, 4K, 16K (in each dimension)

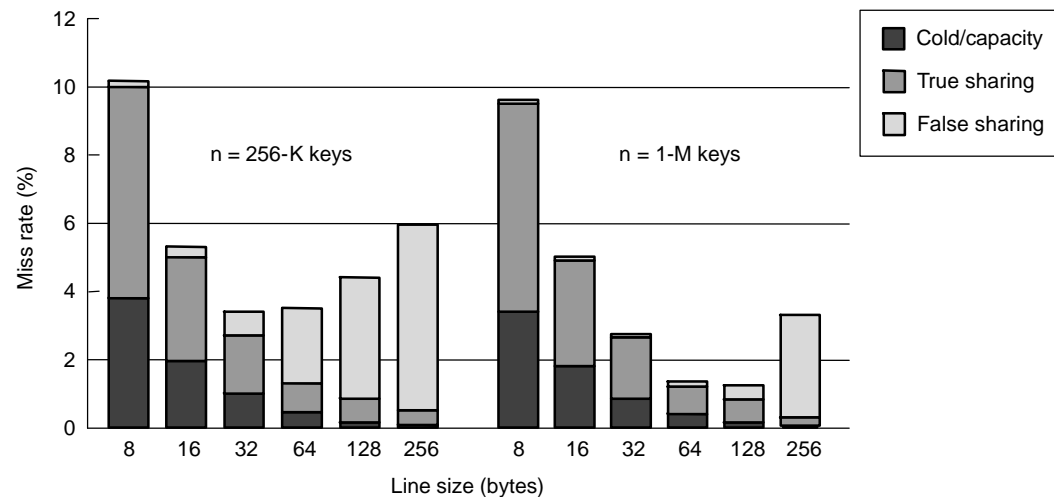
# Steps in Choosing Problem Sizes (contd)

## 5. Use spatial locality and granularity interactions

- E.g., in grid solver, can we distribute data at page granularity in SAS?
  - Affects whether cache misses are satisfied locally or cause comm.
  - With 2D array representation, grid sizes 512, 1K, 2K no, 4K, 16K yes
  - For 4-D array representation, yes except for very small problems
- So no need to expand choices for this reason

## More stark example: false sharing in Radix sort

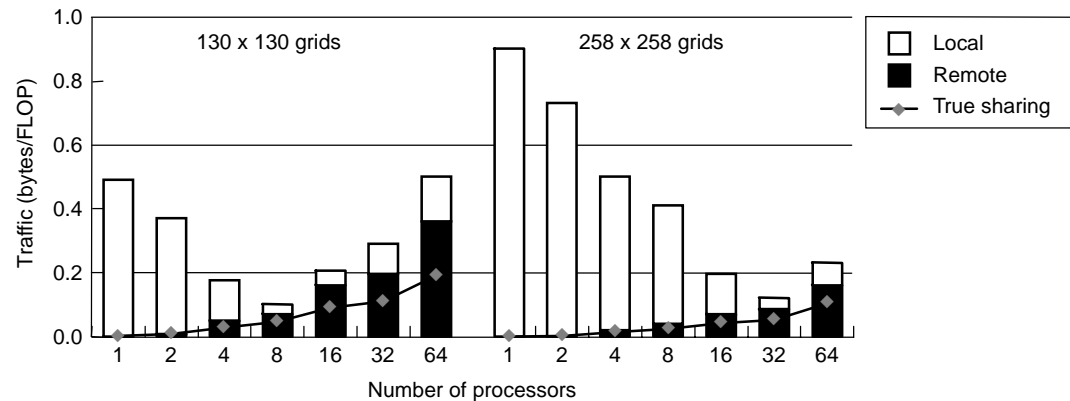
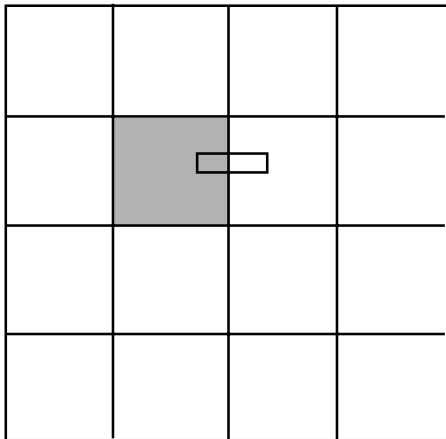
- Becomes a problem when cache line size exceeds  $n/(r*p)$  for radix  $r$



Many applications don't display strong dependence (e.g. Barnes-Hut)

# Summary Example 1: Ocean

$n$ -by- $n$  grid  
with  
 $p$  processors  
(computation  
like grid  
solver)



$n/p$  is large  $\Rightarrow$

- Low communication to computation ratio
- Good spatial locality with large cache lines
- Data distribution and false sharing not problems even with 2-d array
- Working set doesn't fit in cache; high local capacity miss rate.

$n/p$  is small  $\Rightarrow$

- High communication to computation ratio
- Spatial locality may be poor; false-sharing may be a problem
- Working set fits in cache; low capacity miss rate.

*e.g. Shouldn't make conclusions about spatial locality based only on small problems, particularly if these are not very representative.*

# Summary Example 2: Barnes-Hut

Large problem size (large  $n$  or small  $\theta$ ) relative to  $p$   $\Rightarrow$  most of the time is in force-computation phase

- Good load balance
- Low communication to computation ratio

Small problem size relative to  $p$   $\Rightarrow$  much time in tree building phase

- Poor load balance, large synchronization wait times
- High communication to computation ratio

*Must choose range of problem sizes based on workload properties*

- Many issues and no universal rules, but these guidelines help
- Usually number of problem sizes needed, since few key thresholds

# Varying $p$ on a Given Machine

Already know how to scale problem size with  $p$

Issue: What are starting problem sizes for scaling?

- Could use three sizes (small, medium, large) from fixed- $p$  evaluation above and start from there
- Or pick three sizes on uniprocessor and start from there
  - small: fits in cache on uniprocessor, and significant c-to-c ratio
  - large: close to filling memory on uniprocessor
    - working set doesn't fit in cache on uniprocessor, if this is realistic
  - medium: somewhere in between, but significant execution time

How to evaluate PC scaling with a large problem?

- Doesn't fit on uniprocessor or may give highly superlinear speedups
- Measure speedups relative to small fixed  $p$  instead of  $p=1$

# Metrics for Comparing Machines

Both cost and performance are important (as is effort)

- For a fixed machine as well as how they scale
- E.g. if speedup increases less than linearly, may still be very cost effective if cost to run the program scales sublinearly too
- Some measure of “cost-performance” is most useful

But cost is difficult to get a handle on

- Depends on market and volume, not just on hardware/effort

Also, cost and performance can be measured independently

Focus here on measuring performance

Many metrics used for performance

- Based on absolute performance, speedup, rate, utilization, size ...
- Some important and should always be presented
- Others should be used only very carefully, if at all
- Let’s examine some ...

# Absolute Performance

Wall-clock is better than CPU user time

- CPU time does not record time that a process is blocked waiting

Doesn't help understand bottlenecks in time-sharing situations

- But neither does CPU user time

What matters is execution time till last process completes

- Not average time over processes

Best for understanding performance is breakdowns of execution time

- Broken down into components as discussed earlier (busy, data ...)

Execution time must be reported as function of problem size and number of processors

# Speedup

Recall  $Speedup_N(p) = Performance_N(p) / Performance_N(1)$

What is  $Performance(1)$ ?

1. Parallel program on one processor of parallel machine?
  2. Same sequential algorithm on one processor of parallel machine?
  3. “Best” sequential program on one processor of parallel machine?
  4. “Best” sequential program on agreed-upon standard machine?
3. is more honest than 1. or 2. for users
- 2. may be okay for architects to understand parallel performance
4. evaluates uniprocessor performance of machine as well
- Similar to absolute performance

# Processing Rates

Popular to measure computer operations per unit time

- MFLOPS, MIPS
- As opposed to operations that have meaning at application level

Neither good for comparing machines

- Can be artificially inflated
  - Worse algorithm with greater FLOP rate, or even add useless cheap ops
- Who cares about FLOPS anyway?
- Different floating point ops (add, mul, ...) take different time
- Burdened with legacy of misuse

Can use independently known no. of operations as work measure

- Then rate no different from measuring execution time

*Okay for comparing measured hardware performance with peak for a given implementation of a benchmark, but not for much else*

# Resource Utilization

Architects often measure how well resources are utilized

- E.g. processor utilization, memory...

Not a useful metric for a user

- Can be artificially inflated
- Looks better for slower, less efficient resources

May be useful to architect to determine machine bottlenecks/balance

- But not useful for measuring performance or comparing systems

# Metrics based on Problem Size

E.g. Smallest problem size needed to achieve given *parallel efficiency* (parallel efficiency = speedup/ $p$ )

- Motivation: everything depends on problem size, and smaller problems have more parallel overheads
- Distinguish comm. architectures by ability to run smaller problems
- Introduces another scaling model: *efficiency-constrained scaling*

## Caveats

- Sometimes larger problem has worse parallel efficiency
  - Working sets have nonlocal data, and may not fit for large problems
- Small problems may not stress local memory system

Often useful for understanding improvements in comm. Architecture

- Especially useful when results depend greatly on problem size
- But not a generally applicable performance measure

# Percentage Improvement in Performance

Often used to evaluate benefit of an architectural feature

Dangerous without also mentioning original parallel performance

- Improving speedup from 400 to 800 on 1000 processor system is different than improving from 1.1 to 2.2
- Larger problems, which get better base speedup, may not see so much improvement

## Summary of metrics

- For user: absolute performance
- For architect: absolute performance as well as speedup
  - any study should present both
  - size-based metrics useful for concisely including problem size effect
- Other metrics useful for specialized reasons, usually to architect
  - but must be careful when using, and only in conjunction with above

# Presenting Results

Beware peak values

- Never obtained in practice
  - Peak = “guaranteed not to exceed”
- Not meaningful for a user who wants to run an application

Even single measure of “sustained” performance is not really useful

- Sustained on what benchmark?

Averages over benchmarks are not very useful

- Behavior is too dependent on benchmark, so average obscures

Report performance per benchmark

- Interpreting results needs understanding of benchmark

Must specify problem and machine configuration

- Can make dramatic difference to results; e.g. working set fits or not

# “Twelve Ways to Fool the Masses”

Compare 32-bit results to others' 64-bit results

Present inner kernel results instead of whole application

Use assembly coding and compare with others' Fortran or C codes

Scale problem size with number of processors, but don't tell

Quote performance results linearly projected to a full system

Compare with scalar, unoptimized, uniprocessor results on CRAYs

Compare with old code on obsolete system

Use parallel code as base instead of best sequential one

Quote performance in terms of utilization, speedups/peak MFLOPS/\$

Use inefficient algorithms to get high MFLOPS rates

Measure parallel times on dedicated sys, but uniprocessor on busy sys

Show pretty pictures and videos, but don't talk about performance

# Some Important Observations

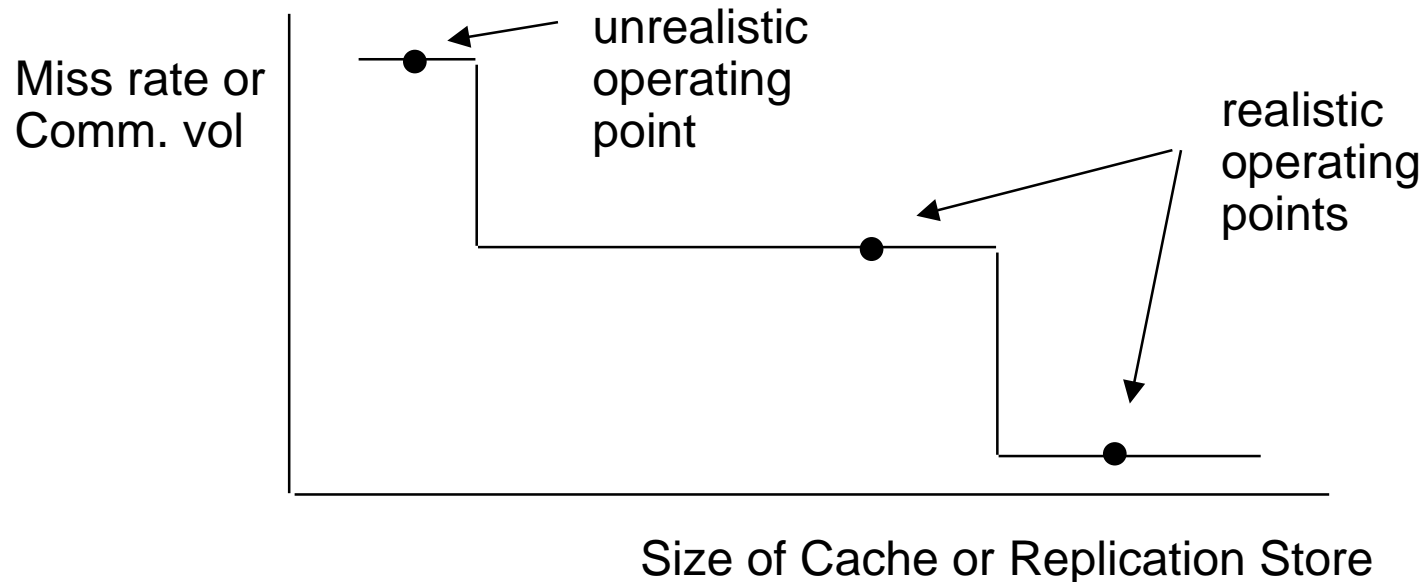
In addition to assignment/orchestration, many important properties of a parallel program depend on:

- Application parameters and number of processors
- Working sets and cache/replication size

*Should cover realistic regimes of operation*

# Operating Points Based on Working Sets

Many applications have a hierarchy of working sets:



- A working set may consist of local and/or nonlocal data
  - not fitting it may dramatically increase local miss rate or even communication
- Some working sets scale with application parameters and  $p$ , some don't
- Some operating points are realistic, some aren't
  - *operating point = f (cache/replication size, application parameters,  $p$ )*

# Evaluating an Idea or Tradeoff

Typically many things change from one generation to next

Building prototypes for evaluation is too expensive

Build a simulator

Case I: Want to examine in the context of a given

- Can assume technological and architectural parameters
  - Simulate with feature turned off and turned on to examine impact
  - Perhaps examine sensitivity to some parameters that were fixed
- Building accurate simulators is complex
  - Contention difficult to model correctly
  - Processors becoming increasingly complex themselves

Case II: Want to examine benefit of idea in a more general context

- Now machine parameters also variable
  - Various sizes, granularities and organizations, performance characteristics

# Multiprocessor Simulation

Simulation runs on a uniprocessor (can be parallelized too)

- Simulated processes are interleaved on the processor

Two parts to a simulator:

- Reference generator: plays role of simulated processors
  - And schedules simulated processes based on *simulated time*
- Simulator of extended memory hierarchy
  - Simulates operations (references, commands) issued by reference generator

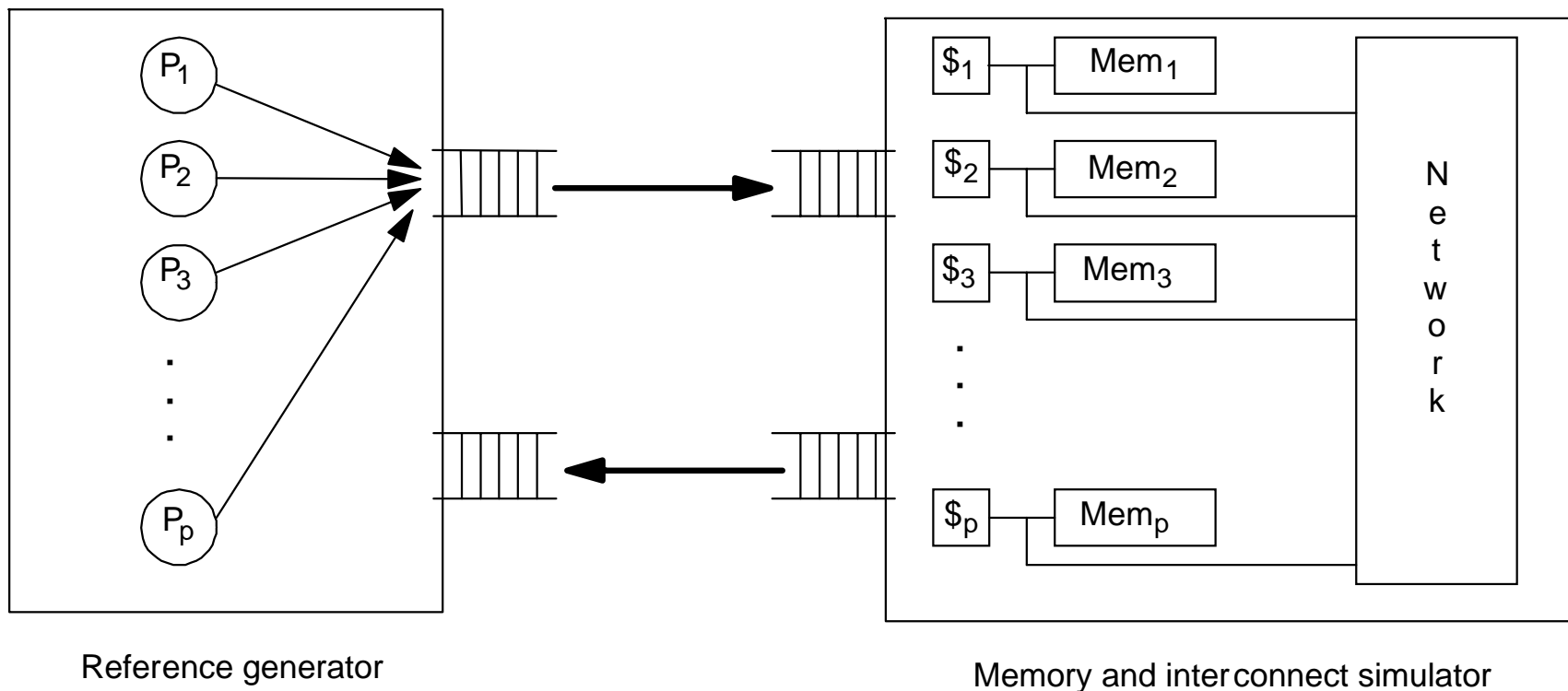
Coupling or information flow between the two parts varies

- Trace-driven simulation: from generator to simulator
- Execution-driven simulation: in both directions (more accurate)

Simulator keeps track of simulated time and detailed statistics

# Execution-driven Simulation

Memory hierarchy simulator returns simulated time information to reference generator, which is used to schedule simulated processes



# Difficulties in Simulation-based Evaluation

Two major problems, beyond accuracy and reliability:

- Cost of simulation (in time and memory)
  - cannot simulate the problem/machine sizes we care about
  - have to use scaled down problem and machine sizes
    - *how to scale down and stay representative?*
- Huge design space
  - application parameters (as before)
  - machine parameters (depending on generality of evaluation context)
    - number of processors
    - cache/replication size
    - associativity
    - granularities of allocation, transfer, coherence
    - communication parameters (latency, bandwidth, occupancies)
  - cost of simulation makes it all the more critical to prune the space

# Scaling Down Parameters for Simulation

Want scaled-down machine running scaled-down problem to be representative of full-sized scenario

- No good formulas exist
- But very important since reality of most evaluation
- Should understand limitations and guidelines to avoid pitfalls

First examine scaling down problem size and no. of processors

Then lower-level machine parameters

Focus on cache-coherent SAS for concreteness

# Scaling Down Problem Parameters

Some parameters don't affect parallel performance much, but do affect runtime, and can be scaled down

- Common example is no. of time-steps in many scientific applications
  - need a few to allow settling down, but don't need more
  - may need to omit cold-start when recording time and statistics
- First look for such parameters
- Others can be scaled according to earlier scaling arguments

But many application parameters affect key characteristics

Scaling them down requires scaling down no. of processors too

- Otherwise can obtain highly unrepresentative behavior

# Difficulties in Scaling $N, p$ Representatively

Many goals, difficult individually and often impossible so to reconcile

Want to preserve many aspects of full-scale scenario

- Distribution of time in different phases
- Key behavioral characteristics
- Scaling relationships among application parameters
- Contention and communication parameters

Can't really hope for full representativeness, but can

- Cover range of realistic operating points
- Avoid unrealistic scenarios
- Gain insights and estimates of performance

# Scaling Down Other Machine Parameters

Often necessary when scaling down problem size

- E.g. may not represent working set not fitting if cache not scaled

More difficult to do with confidence

- *Cache/replication size*: guide by scaling of working sets, not data set
- *Associativity and Granularities*: more difficult
  - should try to keep unchanged since hard to predict effects, but ...
  - greater impact with scaled-down application and system parameters
  - difficult to find good solutions for both communication and local access

Solutions and confidence levels are application-specific

- Require detailed understanding of application-system interactions

Should try to use as realistic sizes as possible

- Use guidelines to cover key operating points, and extrapolate with caution

# Dealing with the Parameter Space

## Steps in an evaluation study

- Determine which parameters are relevant to evaluation
- Identify values of interest for them
  - context of evaluation may be restricted
- Analyze effects where possible
- Look for knees and flat regions to prune where possible
- Understand growth rate of characteristic with parameter
- Perform sensitivity analysis where necessary

# An Example Evaluation

Goal of study: To determine the value of adding a block transfer facility to a cache-coherent SAS machine with distributed memory

Workloads: Choose at least some that have communication that is amenable to block transfer (e.g. grid solver)

Choosing parameters is more difficult. 3 goals:

- Avoid unrealistic execution characteristics
- Obtain good coverage of realistic characteristics
- Prune the parameter space based on
  - goals of study
  - restrictions imposed by technology or assumptions
  - understanding of parameter interactions

Let's use equation solver as example

# Choosing Parameters

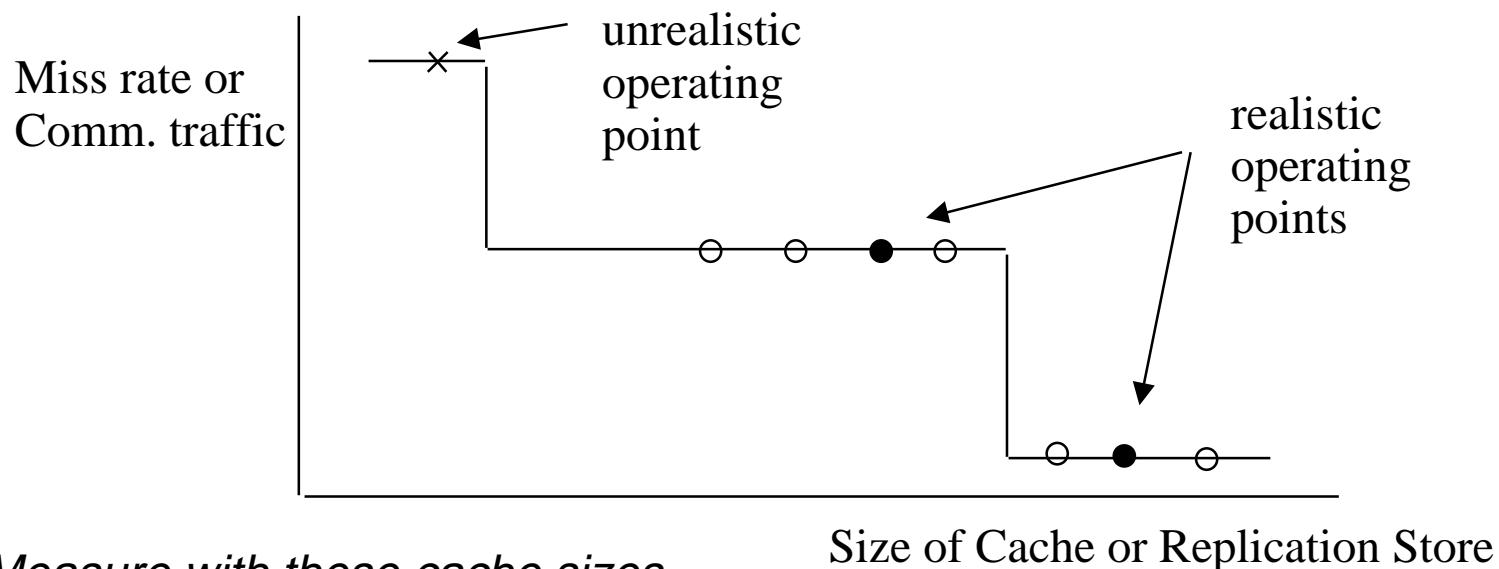
## Problem size and number of processors

- Use inherent characteristics considerations as discussed earlier
- For example, low c-to-c ratio will not allow block transfer to help much
- Suppose one size chosen is 514-by-514 grid with 16 processors

## Cache/Replication Size

- Choose based on knowledge of working set curve
- Choosing cache sizes for given problem and machine size analogous to choosing problem sizes for given cache and machine size, discussed
- Whether or not working set fits affects block transfer benefits greatly
  - if local data, not fitting makes communication relatively less important
  - If nonlocal, can increase artifactual comm. So BT has more opportunity
- Sharp knees in working set curve can help prune space (next slide)
  - Knees can be determined by analysis or by very simple simulation

# Example of Pruning using Knees



- *Measure with these cache sizes*
- *Don't measure with these cache sizes*

But be careful: applicability depends on what is being evaluated

- what if miss rate isn't all that matters from cache (see update/invalidate protocols later)

If growth rate can be predicted, can prune for other  $n, p, \dots$  too

Often knees are not sharp, in which case use sensitivity analysis

# Choosing Parameters (contd.)

Cache block size: issues more detailed

- Long cache blocks behave like small block transfers already
- When spatial locality is good, explicit block transfer less important
- When spatial locality is bad
  - waste bandwidth in read-write communication
  - but also in block transfer IF implemented on top of cache line transfers
  - block transfer itself increases bandwidth needs (same comm. in less time)
  - so it may hurt rather than help if spatial locality bad and implemented on top of cache line transfers, if bandwidth is limited
- Fortunately, range of interesting line sizes is limited
  - if thresholds occur, as in Radix sorting, must cover both sides

Associativity

- Effects difficult to predict, but range of associativity usually small
- Be careful about using direct-mapped lowest-level caches

# Choosing Parameters (contd.)

Overhead, network delay, assist occupancy, network bandwidth

- Higher overhead for cache miss: greater amortization with BT
  - unless BT overhead swamps it out
- Higher network delay, greater benefit of BT amortization
  - no knees in effects of delay, so choose a few in the range of interest
- Network bandwidth is a saturation effect:
  - once amply adequate, more doesn't help; if low, then can be very bad
  - so pick one that is less than the knee, one near it, and one much greater
  - Take burstiness into account when choosing (average needs may mislead)

## Revisiting choices

- Values of earlier parameters may have be revised based on interactions with those chosen later
- E.g. choosing direct-mapped cache may require choosing larger caches

# Summary of Evaluating a Tradeoff

Results of a study can be misleading if space not covered well

- Sound methodology and understanding interactions is critical

While complex, many parameters can be reasoned about at high level

- Independent of lower-level machine details
- Especially: problem parameters, no. of processors, relationship between working sets and cache/replication size
- Benchmark suites can provide and characterize these so users needn't

Important to look for knees and flat regions in interactions

- Both for coverage and for pruning the design space

High-level goals and constraints of a study can also help a lot

# Illustrating Workload Characterization

We'll use parallel workloads throughout to illustrate tradeoffs

Illustrate characterization with those for a cache-coherent SAS

- Six parallel applications running in batch mode
  - taken from SPLASH-2 suite
- One multiprogrammed workload which includes OS activity
- Small number, but chosen to exhibit a wide range of behaviors

Three workloads from case-study applications discussed earlier

- Ocean, Barnes-Hut, Raytrace

Briefly describe the four new workloads:

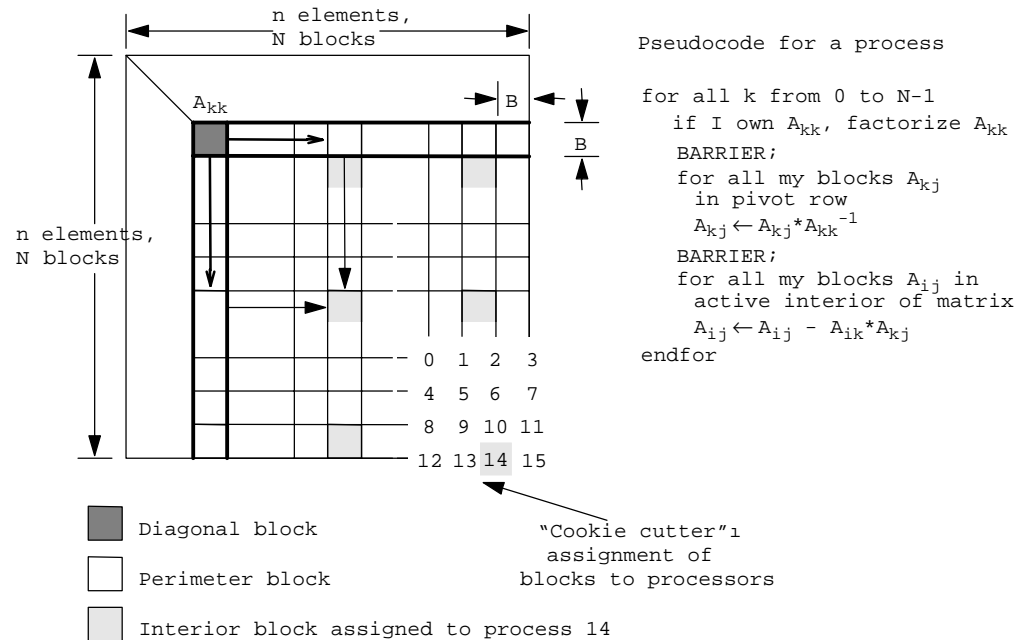
- LU, Radix, Radiosity, Multiprog

Illustrate methodologically relevant characteristics

- Will use to choose parameters for later illustrative evaluations

# LU: Dense Matrix Factorization

Factorize matrix  $A$  into lower- and upper-triangular matrices:  $A = LU$



- Blocked for reuse in the cache:  $B^3$  ops on  $B^2$  data for each block “task”
  - Tradeoffs in block size
- Scatter assignment of blocks improves load balance in later stages
- Good temporal and spatial locality (4-D arrays), low communication
- Only barrier synchronization (conservative but simple)

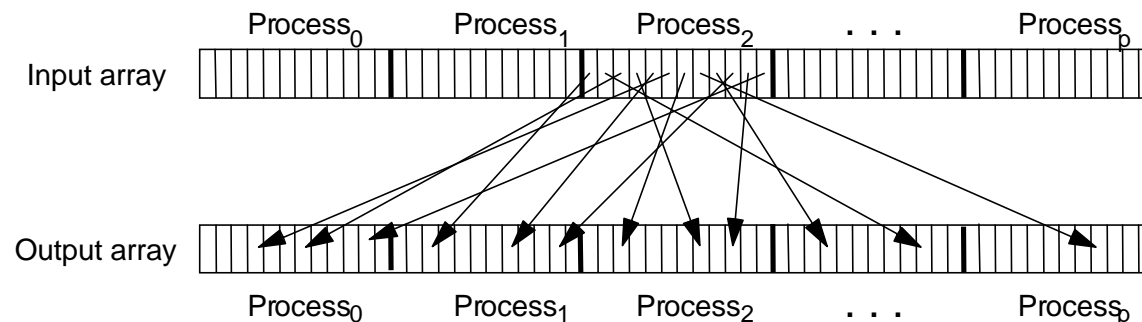
# Radix

Sort  $b$ -bit integer keys, using radix  $r$  (i.e.  $b/r$  phases using  $r$  bits each)

- Each phase reads from input array, writes to output array, alternating

Three steps within each phase

- Build local histogram of key frequencies
- Combine local histograms into global histogram (prefix computation)
- Permute from input to output array based on histogram values



- All-to-all personalized comm., highly scattered remote writes
  - Poor spatial locality
- Locks and/or pt.-to-pt. events in second step and barriers between steps

# Radiosity: Global Illumination in a Scene

Hierarchical division of polygons starting from set of initial ones

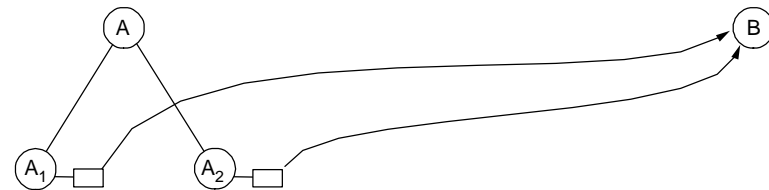
- Based on interactions among polygons/patches, using interaction lists

- A lot of time spent computing visibility between patch pairs
- Parallelism across polygons, patches (used here), or interactions
- Highly irregular and unpredictable access and sharing patterns
- Dynamic load balancing with task queues; fine-grained locking

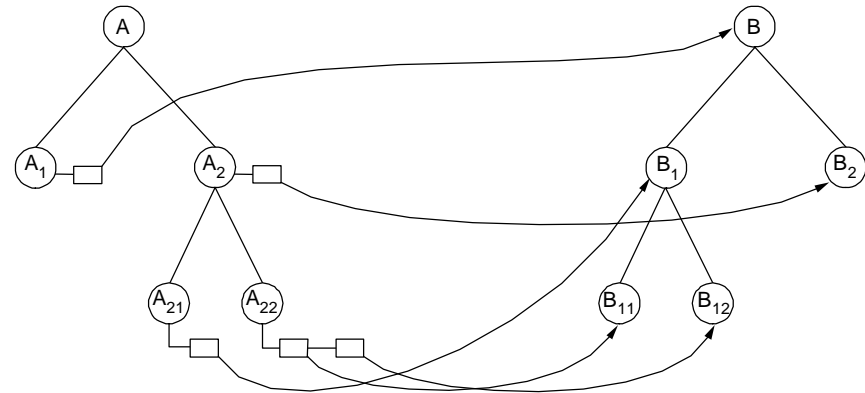
(1) Before refinement



(2) After the first refinement



(3) After three more refinements:  $A_2$  subdivides B; then  $A_2$  is subdivided due to  $B_{2,1}$ ; then  $A_{22}$  subdivides  $B_1$



# Multiprog: A Simple Mix

## Multiprogrammed sequential programs

- Multiprocessor used as throughput server, with single OS image
- Very popular use of small-scale, shared memory multiprocessors
- OS is significant component of execution (itself a parallel application)
- Two UNIX file compress jobs
- Two parallel compilations (pmake)
- OS is a version of SGI UNIX (IRIX version 5.2)

# Characteristics: Data Access and Synch.

Using default problem sizes (at small end of realistic for 64-p system)

Table 4.1 General Statistics about Application Programs

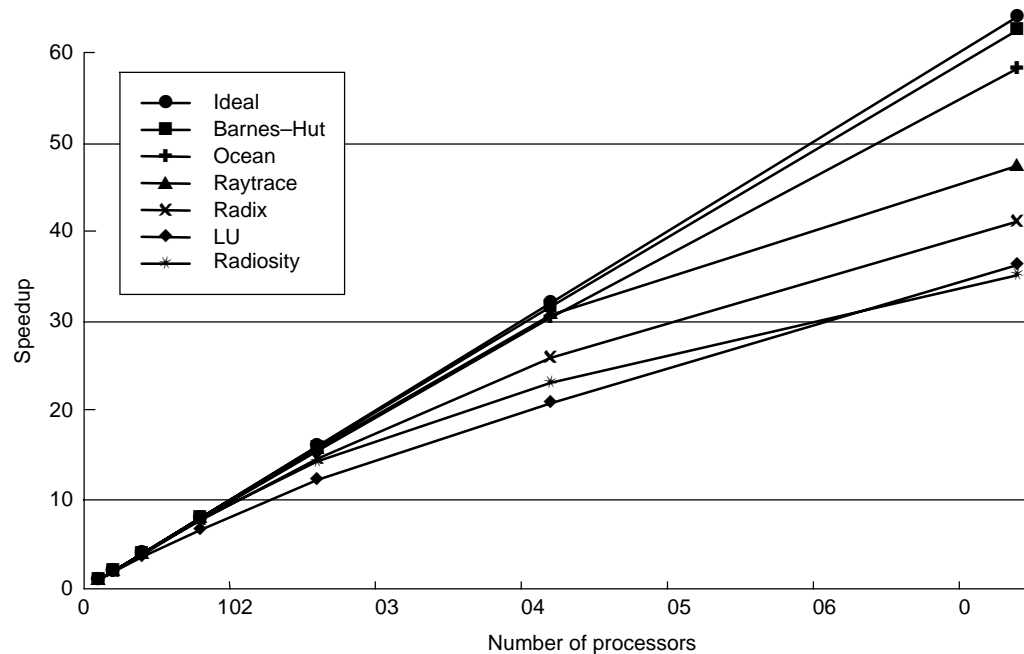
Application	Input Data Set	Total Instructions (M)	Total FLOPS (M)	Total References (M)	Total Reads (M)	Total Writes (M)	Shared Reads (M)	Shared Writes (M)	Barriers	Locks
LU	512× 512 matrix 16 × 16 blocks	489.52	92.20	151.07	103.09	47.99	92.79	44.74	66	0
Ocean	258× 258 grids tolerance = 10 <sup>-7</sup> 4 time-steps	376.51	101.54	99.70	81.16	18.54	76.95	16.97	364	1,296
Barnes-Hut	16-K particles θ = 1.0 3 time-steps	2,002.74	239.24	720.13	406.84	313.29	225.04	93.23	7	34,516
Radix	256-K points radix = 1,024	84.62	—	14.19	7.81	6.38	3.61	2.18	11	16
Raytrace	Car scene	833.35	—	290.35	210.03	80.31	161.10	22.35	0	94,456
Radiosity	Room scene	2,297.19	—	769.56	486.84	282.72	249.67	21.88	10	210,485
Multiprog: User	SGI IRIX 5.2, two pmake+ two compress jobs	1,296.43	—	500.22	350.42	149.80	—	—	—	—
Multiprog: Kernel		668.10	—	212.58	178.14	34.44	—	—	—	621,505

For the parallel programs, shared reads and writes simply refer to all nonstack references issued by the application processes. Such references do not necessarily point to data that is truly shared by multiple processes. The Multiprog workload is not a parallel application, so it does not access shared data. A dash in a table entry means that this measurement is not applicable to that application (e.g., Radix has no floating-point operations). (M) denotes that measurement in that column is in millions.

•Operation distributions vary widely across workloads (coverage)

# Concurrency and Load Balance

Speedups on PRAM with default problem size



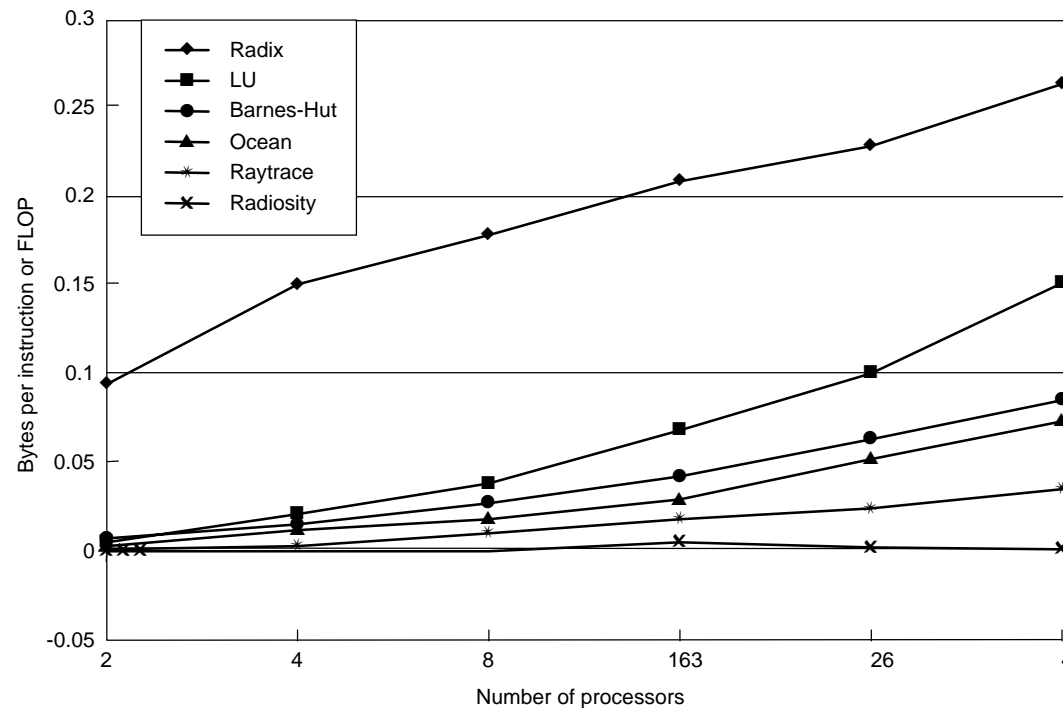
- All speed up well: Radiosity, LU and Radix are worst
  - Different reasons, alleviated by larger data sets so valid for big systems too
  - Applications have good concurrency (satisfy another criterion)

# Communication to Computation Ratio

Simulate infinite caches with one-word block to eliminate artifactual

- Measure as bytes per instruction or bytes per FLOP

Under PC scaling with default problem size



- Average bw need generally small, except for Radix (so watch out for it)
  - well-optimized, but burstiness may make others demand more bw too
- Scaling trend with  $p$  different across applications

# Growth Rates of Comm-to-comp Ratio

Table 4.1 Growth Rates of Inherent Communication-to-Computation Ratio

Application	Growth Rate
LU	$\sqrt{PD/S}$
Ocean	$\sqrt{PD/S}$
Barnes-Hut	Approximately $\sqrt{PD/S}$
Radiosity	Unpredictable
Radix	$(P-1)P$
Raytrace	Unpredictable

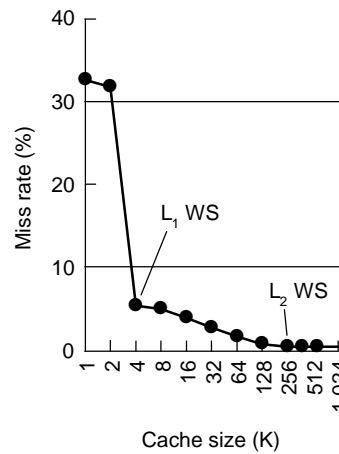
DS is the data set size (in bytes, say), and P is the number of processes.

- DS is data set size; rate depends on other parameters as well
- Depends a lot on DS in some cases (Ocean, LU), less in others
  - constant factors can be more important than growth rates for real systems
- Artifactual communication is whole different story
  - later, in context of architecture types

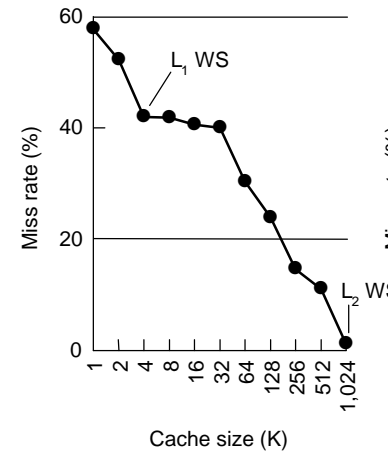
# Working Set Curves (Default Config.)

Measure using fully associative LRU caches with 8-byte cache block

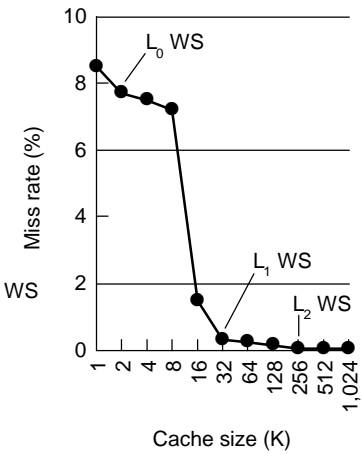
- Small associativity can affect whether working set fits or not
- Key working sets are well-defined in most cases



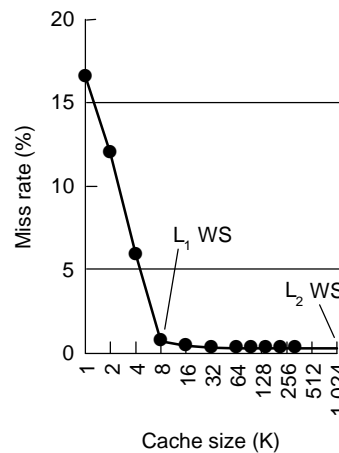
(a) LU



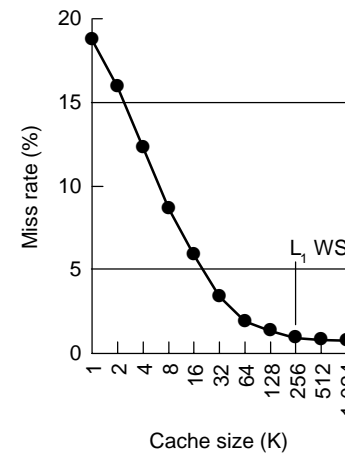
(b) Ocean



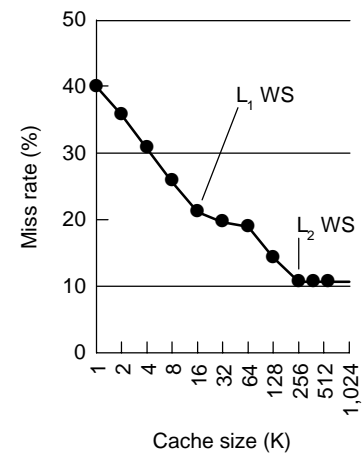
(c) Barnes-Hut



(d) Radiosity



(e) Raytrace



(f) Radix

# Working Set Growth Rates

How they scale, whether important, and realistic to fit/not?

Table 4.1 Important Working Sets and Their Growth Rates for the SPLASH-2 Suite

Program	Working Set 1	Growth Rate	Important?	Realistic Not to Fit in Cache?	Working Set 2	Growth Rate	Important?	Realistic Not to Fit in Cache?
LU	One block	Fixed	Yes	No	Partition of DS	DS/P	No	Yes
Ocean	A few subrows	$\sqrt{PD} \sqrt{S}$	Yes	No	Partition of DS	DS/P	Yes	Yes
Barnes-Hut	Tree data for 1 body	$(\log DS) / \theta^2$	Yes	No	Partition of DS	DS/P	No	Yes
Radiosity	BSP tree	$\log(\text{polygons})$	Yes	No	Unstructured	Unstructured	No	Yes
Radix	Histogram	Radix	Yes	No	Partition of DS	DS/P	Yes	Yes
Raytrace	Scene and grid data reused across rays	Unstructured	Yes	Yes	—	—	—	—

DS represents the data set size, and  $P$  is the number of processes.

- Ocean, Radix and Raytrace realistic not to fit in modern L2 cache
  - So should represent that situation as well for them

# Concluding Remarks

Guidelines for workload-driven evaluation

- Choosing workloads, scaling, dealing with large space, metrics
- Challenging task: outlined steps, pitfalls, limitations

Requires understanding application-architecture interactions

- Benchmark suites can provide the rules and characterization

Described some of the workloads to be used later

Now on firm footing to proceed to core architecture ...