Must Parallel Programming be Hard?

How to Live with and Survive Multi-Core

Marc Snir

PARALLEL@ILLINOIS

www.parallel.illinois.edu

PARALLEL@ILLINOIS

Illiac

UPCRC

UPCRC Illinois
Universal Parallel Computing Research Center

Gigascale System Research Center

Petascale computing

Institute for Advanced Computing Applications and Technologies

GSRC

BLUE WATERS
PETASCALE COMPUTING SYSTEM

concurrent

Cloud Computing Testbed (CCT)

Multi-Core: All Computers Are Now Parallel

- We continue to have more transistors per chip
  - Moore's Law
- Cannot continue increasing clock cycle
  - Power
- Cannot continue increasing single thread performance
  - Diminishing returns on added circuitry
- ➡ Transistors are used to populate chips with an increasing numbers of cores

The Computer Economy

- **Old game**: Each new h/w generation provides better user experience (performance) with little/no change in s/w
  - People buy new laptop every 3 years (or less) – good for Intel & Microsoft
- **New game**: Each new chip generation provides better experience *only for applications that run in parallel & scale*
- **Goal**: Better user experience with increased number of cores with little/no code rewrite

The Consequences of Failure

- A very different business model for the IT industry
- A slowdown

HPC wire

PRINT THIS

Click to Print

SAVE THIS | EMAIL THIS | Close

January 08, 2009

Will Multicore Kill the x86?

The hardware and software challenges of multicore/manycore CPUs have been flogged in this publication for a number of years. The assumption was that geek ingenuity would eventually power through the roadblocks. The memory wall problem would yield to innovative hardware architectures, and new software development approaches would make multithreaded computing practical enough for widespread use. But what if that doesn't happen?

Parallel Software is Hard

- Prone to subtle, hard to reproduce bugs
- Much more complex to test
- More complex application mapping to hardware
- Immature development environments
- Lack of trained manpower

Two Hypotheses

A. The development of parallel software is inherently hard
   - Hard to think parallel

B. The development of parallel software need not be (much) harder than the development of sequential software
   - Currently hampered by lack of good programming models, tools, education, etc.
   - Can be cured by suitable investments

Arguments for B

- Some forms of parallel programming are easy
- Most current parallel programming environments were developed to support hard forms of parallel programming (e.g., system code or performance programming)
  - Complex interactions
  - Detailed, low-level resource management; machine-dependent code
- Technologies (and $$) exist to do better

(Some) Parallel Programming is a Child's Play

- Etoys
- "Shared-nothing" programming style:
  - Set of independent objects, each with own program and local state; no shared state
  - Object updates its own state and can read the state of other objects
  - Global clock
- Simple interaction model
- **Deterministic Execution**

What is Determinism?

- Given a sequence of inputs, all executions of the program will have the *same perceived* behavior
  - It depends what *same* is
  - It depends what *perception* is

What Is Same?

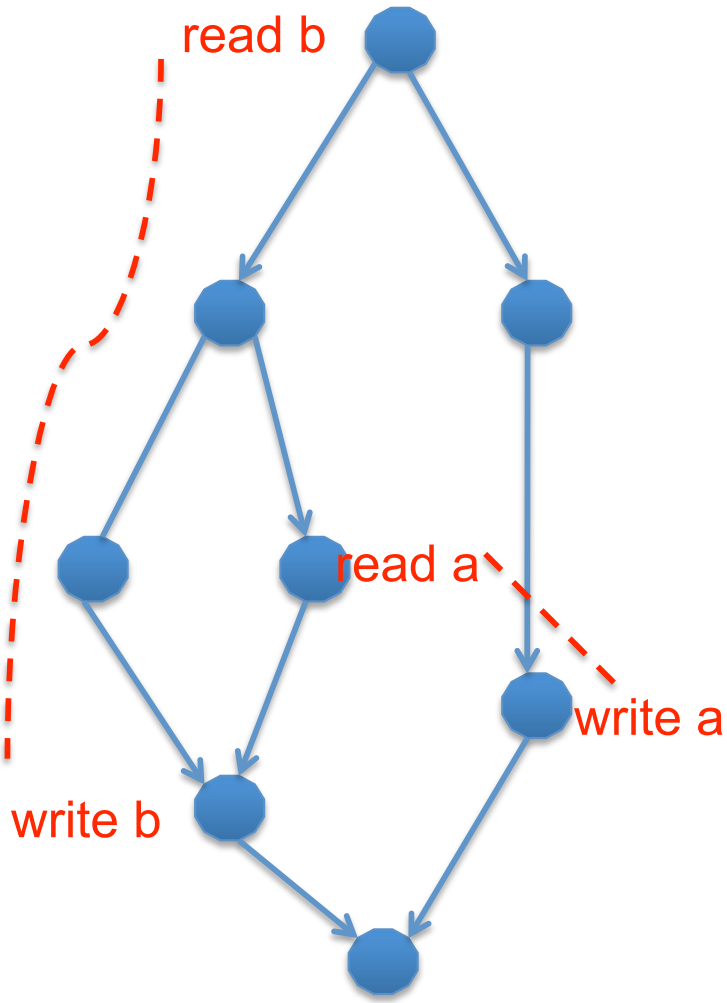
- Same performance?
- Same **ResourceException**?
- Can assume addition is associative and commutative?

- "Same": Equivalence relation on executions

What Can We Observe?

- Outputs
- Non-recovered exceptions
- When debugging, program execution state
  - Assumes operational semantic model

Formalism



- Operations
- Program order
- Conflicts
- Program *deterministic* if program order orders all conflicts
- Deterministic programs have sequential operational semantics

Race-Free is not Enough

```
lock(L);
x++;
unlock(L);
...
```

```
lock(L);
y=x;
unlock(L);
...
```

- Deterministic =

  race-free (all conflicting operations are synchronized)

  + Ordering synchronizations

Why is Determinism Good?

- Easy to test: only one execution path
- Easy to understand: execution equivalent to sequential execution
- Easy to debug
- Easy to incrementally parallelize code
- Can use current tools and methodologies for program development

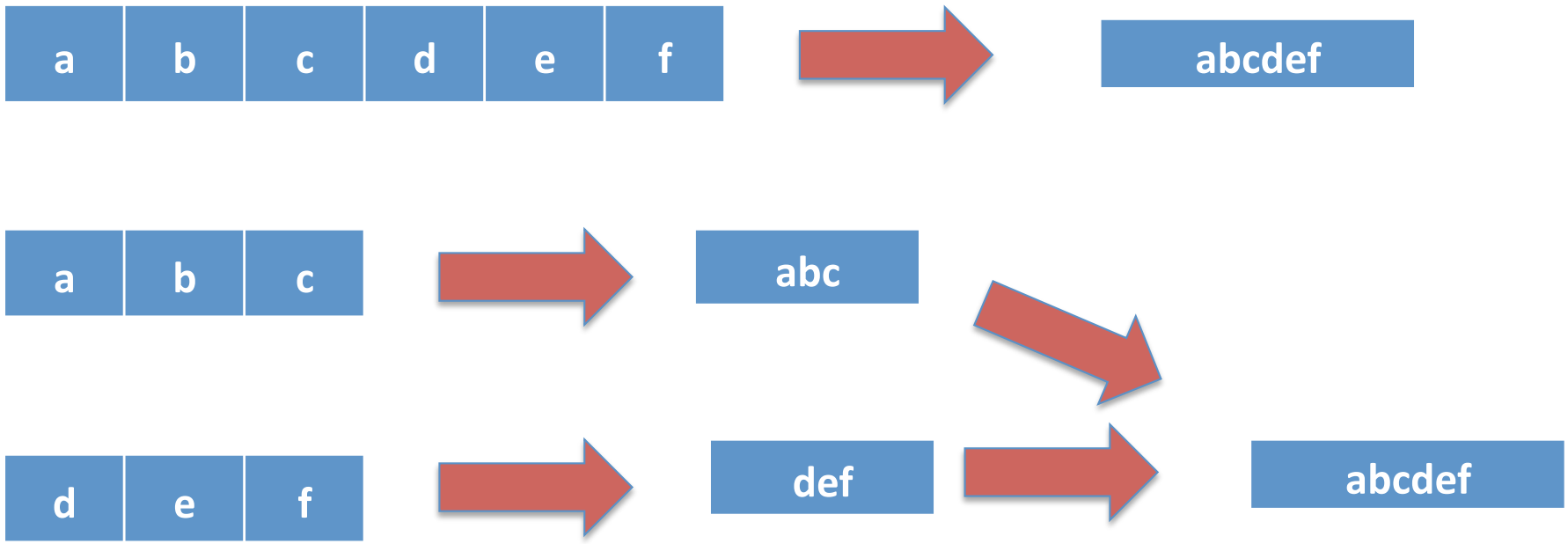
Do We Need Nondeterminism?

- Reactive code: reacts to external events
  - OLTP, OS, GUI
  - Nondeterminism is inherent; inputs are not sequential

How about transformational code?

- Machine-dependent code
- Randomized algorithms

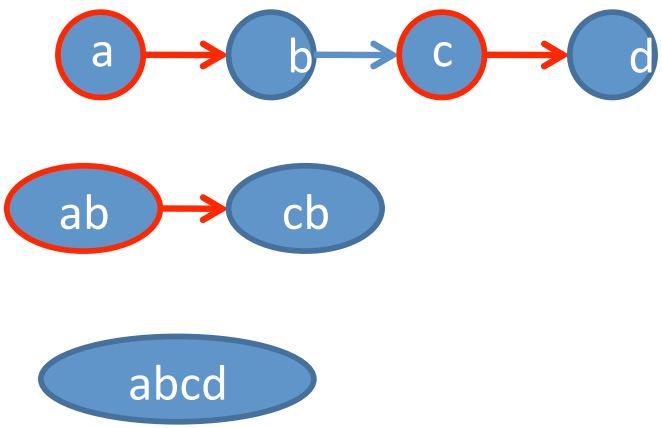
Reduction



- Same set of issues as for optimizing compilers and run-time compilation

Linked List Reduction

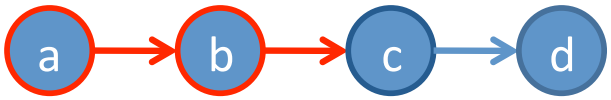
Easy if nodes are stored in contiguous location



Hard if nodes are not sorted

Randomized Linked List Reduction

Pick randomly half of the nodes



Break ties (adjacent nodes) by coin tossing

Each phase reduces # nodes by ~1/4 – optimal within (small) constant factor

Deterministic Linked List Reduction

- Sequence of STOC/FOCS papers by Cole & Vishkin derive a deterministic logarithmic, work optimal algorithm
  - Complex algorithm – asymptotically optimal but not practical
- No proof that nondeterminism (or randomization) is necessary in parallel (transformational) computations
  - Seems to make life easier in some cases (parallel graph algorithms, parallel optimization)

Goal

- Shared-memory language that is deterministic by design and by default
  - unordered conflicts are detected at
    - compile time, if possible
    - run-time, otherwise
- Nondeterministic behavior has to be introduced explicitly using nondeterministic control constructs and is "disciplined"

Hidden Parallelism (1)

- Use conventional sequential language; let compiler + run-time introduce parallelism in a safe manner
  - Parallelizing compilers have had limited success; in particular they are brittle
  - *User has no parallel performance model*

Hidden Parallelism (2)

- Use functional programming language
  - Copying & inefficient use of memory bandwidth
  - Far from established practice
- Use data parallelism (e.g., vector operations)
  - Good, but not enough: need control parallelism

Hidden Parallelism (3)

- Use annotations or "semantically neutral syntax" to declare intended programming model

```
(1) for     i = [lb..ub] loop_body
(2) forall i = [lb..ub] loop_body
```

- *same semantics*
- loop carried dependencies are allowed in first case and disallowed in second case
  - *exception generated if dependency exists*

Run-Time Detection (1)

- Using Thread-Level Speculation
  - Iterates execute in parallel *speculatively.*
  - Variables written are kept in cache; variables read are marked
  - Commit protocol: checks that no variable written by one thread was accessed by another thread during speculative execution
- Can be implemented efficiently in h/w for short threads running concurrently on distinct cores [...,Torrellas 2006]

Run-Time Detection (2)

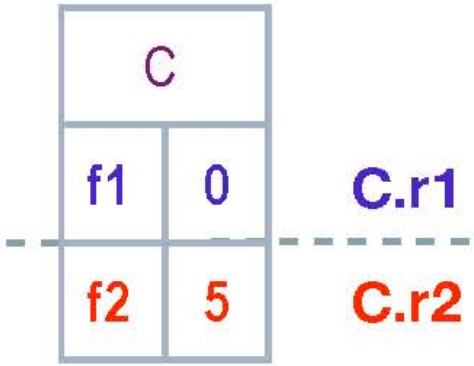
- Good for ensuring that a specific parallel execution does not violate sequential semantics
- Not good enough to ensure that *no* parallel execution will ever violate sequential semantics (i.e., that iterates are independent)

Compile-Time Detection

- Hard for irregular data structures, dynamic partitions, etc.
- Possible approach: allow user to annotate program with type & effect annotations to restrict what can accessed or updated by a task
  - Facilitates compiler analysis (restricts/eliminates run-time checks)
  - User can express implicit knowledge
- Deterministic Parallel Java (*DPJ*) [Bocchino &Adve]

Example: Regions and Effects

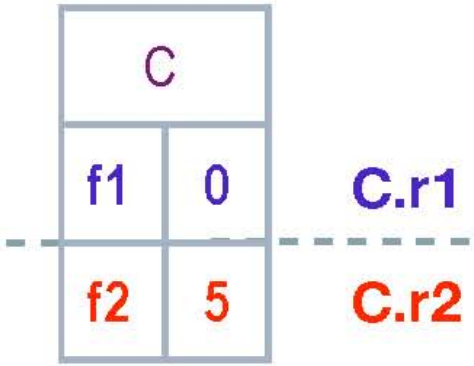
```
class C {
   region r1, r2;
   int f1 in r1;
   int f2 in r2;
   void m1(int x) writes r1 { f1 = x; }
   void m2(int y) writes r2 { f2 = y; }
   void m3(int x, int y) {
      cobegin {
         m1(x);
         m2(y);
      }
   }
}
```



Partitioning the heap

Example: Regions and Effects

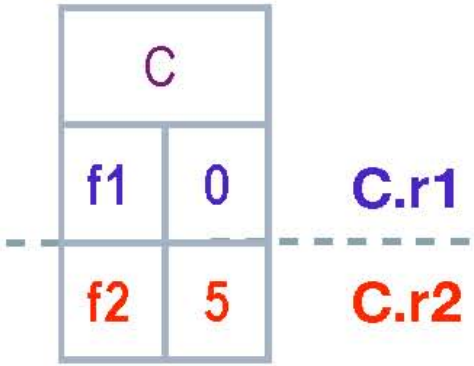
```
class C {
   region r1, r2;
   int f1 in r1;
   int f2 in r2;
   void m1(int x) writes r1 { f1 = x; }
   void m2(int y) writes r2 { f2 = y; }
   void m3(int x, int y) {
      cobegin {
         m1(x);
         m2(y);
      }
   }
}
```



Summarizing method effects

Example: Regions and Effects

```
class C {
   region r1, r2;
   int f1 in r1;
   int f2 in r2;
   void m1(int x) writes r1 { f1 = x; }
   void m2(int y) writes r2 { f2 = y; }
   void m3(int x, int y) {
      cobegin {
         m1(x); // Effect = writes r1
         m2(y); // Effect = writes r2
      }
   }
}
```



Expressing parallelism

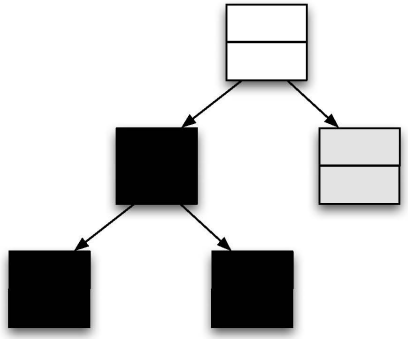
UPCRC Illinois
Universal Parallel Computing Research Center

Using RPLs to Express Nested Shapes

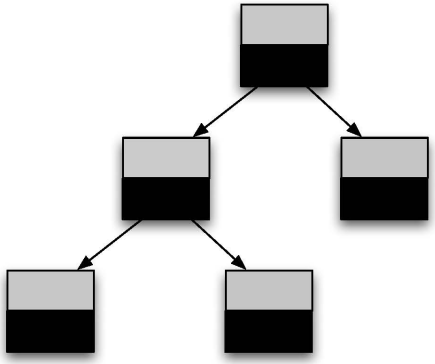
Chain of region names, e.g., *Root : A : B*

Root : A : * **means all regions under *Root : A***

Root : A : * : B **means all regions under *Root : A* ending in *B***



*Root : A : ** vs. *Root : B : **



*Root : * : A* vs. *Root : * : B*

Example: A Tree

```
class Tree<P> {
    region L, R;
    int data in P
    Tree<P:L> left in P:L;
    Tree<P:R> right in P:R;
    …
}
```

P=Root	
data	Root
left	Root:L
right	Root:R

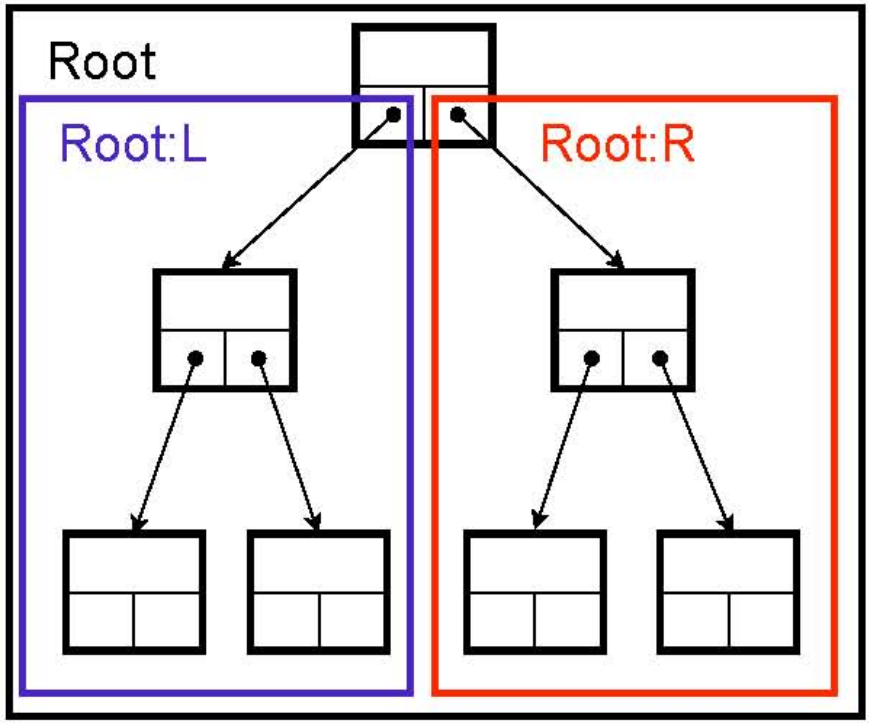
P=Root:L	
data	Root:L
left	Root:L:L
right	Root:L:R

P=Root:R	
data	Root:R
left	Root:R:L
right	Root:R:R

UPCRC Illinois
Universal Parallel Computing Research Center

RPLs: Effects

```
class Tree<P> {
    region L, R;
    int data in P
    Tree<P:L> left in P:L;
    Tree<P:R> right in P:R;
    int increment() writes P:* {
        ++data;
        spawn left.increment();
        right.increment();
    }
    …
}
```



Experiments

Benchmarks

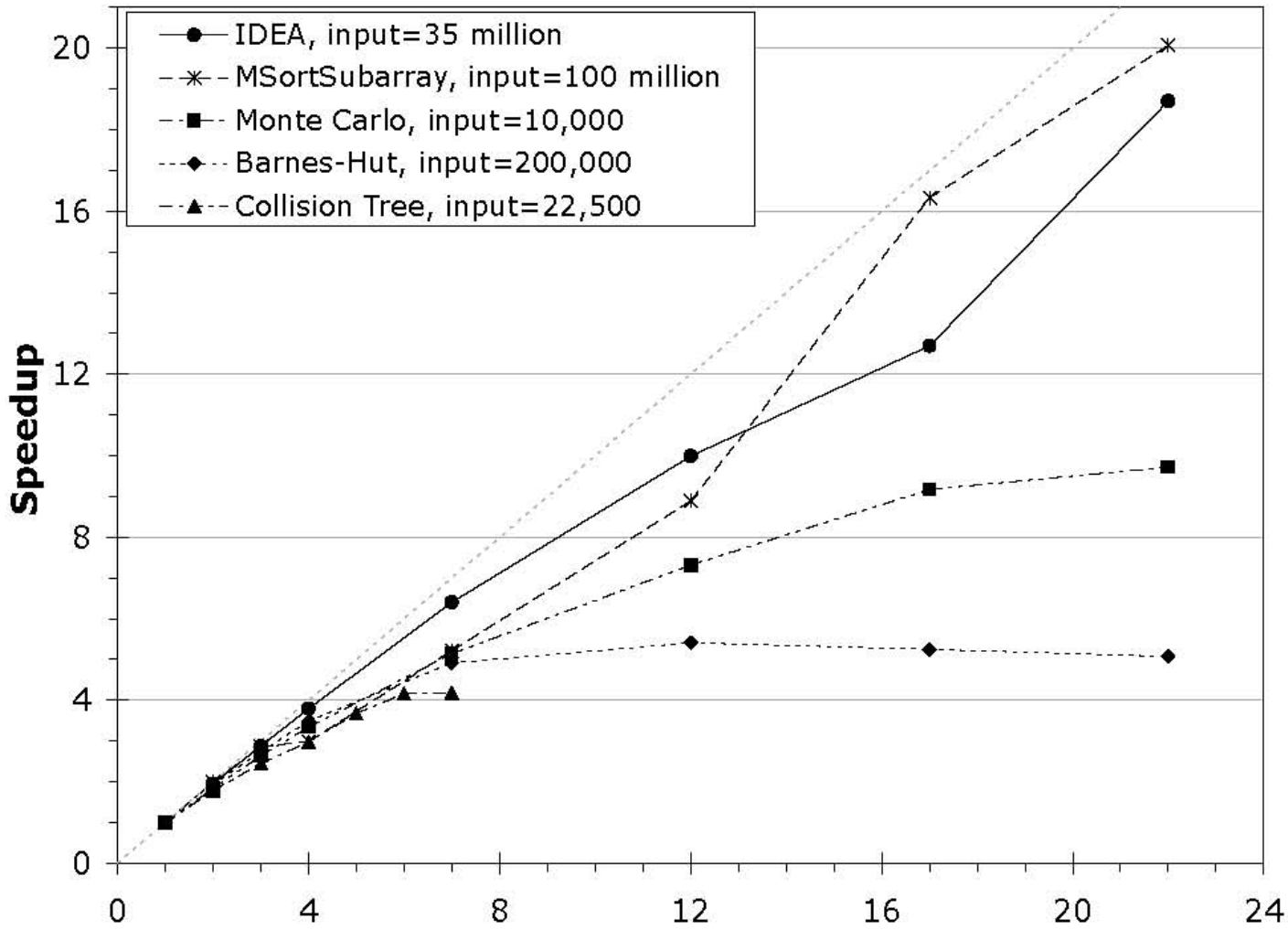
- Merge Sort
- Monte Carlo financial simulation
- IDEA encryption
- Force computation from Barnes-Hut n-body simulation
- Collision Tree from JMonkey graphics engine

Platform

- 4 x 6 core x86 (Dell R900)
- 2GB main memory per core
- 2 x 4 core x86 (Apple Mac Pro) for JMonkey

UPCRC Illinois
Universal Parallel Computing Research Center

Preliminary Performance Results



UPCRC Illinois
Universal Parallel Computing Research Center

Disciplined Nondeterminism

- Linked List Reduction

```
repeat {
  pick p in List where p.next !=
 null;
  p.val += p.next.val;
  p.next = p.next.next;
  } until pick fails;
```

Nondeterministic Iterator

- Equivalent to sequential code
- Construct precisely defines possible serializations
- Analysis indicates that can proceed concurrently with nonadjacent nodes (or proceed speculatively with any number of nodes – [Galois, Pingali])