

Must Parallel Programming be Hard?

How to Live with and Survive Multi-Core

Marc Snir

PARALLEL@ILLINOIS

www.parallel.illinois.edu

PARALLEL@ILLINOIS



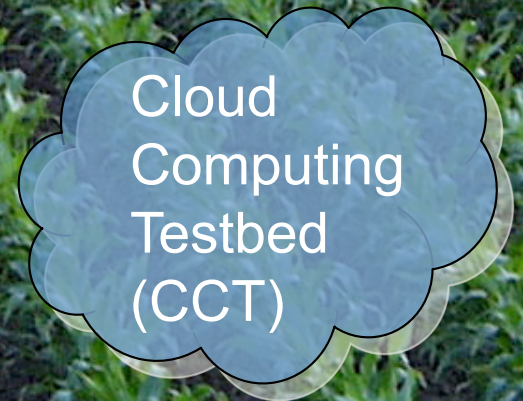
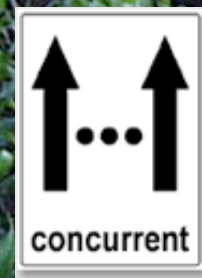
Iliac

UPCRC



Gigascale System
Research Center

Petascale computing




The Computer Economy

- **Moore's law, revisited:** Number of cores per chip increases exponentially
- **New game:** Each new chip generation provides better user experience *only for applications that run in parallel & scale*
- **Goal:** Better user experience with increased number of cores with no code rewrite

The Consequences of Failure

HPCwire

 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?

- Failure means fundamental change in business model of Intel, Microsoft...

What's Parallel Software?

- Code that runs on parallel
 - Php? Rails?
- Code that runs “internally” in parallel
 - SQL?
- Code that expresses a parallel/concurrent algorithm
 - Because concurrency is part of the problem
 - distributed systems, online transaction processing, reactive system
 - Because parallelism is required for performance
 - **This is the only new issue due to multicore: multicore programming is about performance**
 - (actually it is about scalability)

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
- Technologies (and \$\$) exist to do better

(Some) Parallel Programming is a Child's Play

- Etoys
- “Shared-nothing” programming style:
 - Set of objects, each with own program and local state; no shared state
 - Object can update its own state and read the public state of other objects
 - Global clock
- Simple interaction model & deterministic execution
 - **It is easier to code such an application in parallel rather than with sequential code**

Parallel is (Sometimes/Often) Easier: Data Parallel

- point-wise application of pure function

`S.apply (functor)`

- point-wise pure operands

`S1 op S2`

- reduction using pure, associative operand

`S.reduce (op)`

- scan, permute, map...

Parallel (Concurrent) Programming Is Sometime Exceedingly Difficult

- Wait-free synchronization algorithms, Java memory model...
- One gets a PhD thesis for getting it right
- One gets a PhD thesis for showing that the previous guy actually got it wrong
- One gets a PhD thesis for developing formalisms that can be used to show you got it right...

but that nobody uses, because they are too complicated

What Makes Some Parallel Programming Difficult?

- *Lack of isolation*: fine grain, unordered interaction between concurrent modules and program blocks through conflicting accesses to shared variables (**even if race-free**)
 - Effect of the execution of a module depends on other concurrent executions
- *Lack of compositional semantics* (and compositional performance model)
- *Lack of determinism*: state of execution defined by arbitrary interleaving of individual shared memory accesses
- *Lack of safety*: no enforcement of proper synchronization and isolation

Need Structured Parallel Programming

- Need language where correspondence between lexical execution state and dynamic execution state is easy to express (the Dijkstra's argument in "Goto Statement Considered Harmful").
- Need compositional semantics
- Need compositional parallel performance model
 - to express parallel algorithms & control parallel performance
- Need **concurrency-safety**: language-enforced isolation between units that are not supposed to interact

Multicore Programming: Parallelism for Performance

- *Transformational code*: start with input; compute; generate output
- Nondeterminism seldom needed
 - with exceptions such as chaotic relaxation, branch & bound, some parallel graph algorithms
 - when needed, can be highly constrained (limited number of nondeterministic choices)

Example

```
for s in S do f(s);
```

- S is ordered set (or ordered iteration domain)
- iterations executed in order

```
forall s in S do f(s);
```

- Iterations are expected to be independent
- Safety: an exception occurs (compile time? runtime?) otherwise

```
foreach s in S do f(s);
```

- Iterations may execute in arbitrary order
- Safety: atomicity of iterates is enforced

Example (cont.)

- `forall` has same semantic as sequential loop, but parallel performance model
- `foreach` has “sequential semantics”, but choice of sequence is nondeterministic
- `foreach` has parallel performance model: independent iterates execute concurrently (analysis, inspector-executor, speculation)
- Thesis (UPCRC research): safety can be enforced in a parallel OO language with acceptable loss of performance or expressiveness

What Should We Teach

- Foundations:
 - Concurrency (reactive programming) : synchronization, distributed coordination – using abstract models / high-level coordination languages
 - Parallelism (transformational programming): parallel algorithms, parallel complexity – using abstract models / deterministic parallel languages
- Design skills:

System programming, web programming, multicore programming, **performance programming & tuning** – using common languages frameworks and tools
- Need spiral curriculum that integrates both