Programming Big Iron

Jeremy Brown

February 22, 2000

Quick Review

HPF: Fortran 90 plus alignment directives, DOALL, "independent" directive, libraries

NESL: purely functional language; operations on nested (segmented) vectors including nested elementwise invocation; compiles to non-nested vector code

Cilk: C plus spawn, sync, and "inlets"; work-stealing load-balancing; MIMD

pH: Eager, parallel Haskell; side-effects via I- and M-structures only

Java Remote Method Invocation (RMI)

Features:

- "remote" object explicitly placed
- remote objects operated on via method calls
- control-flow transfers to object's host machine
- non-remote arguments/return values are deep-copied

Jade
(Martin Rinard)

C-like language with strictly sequential semantics

User specifies which shared objects a code-segment will read and/or write

Runtime system invokes non-conflicting segments in parallel

Specifications are dynamically verified!

Data-centric parallelism: Elementwise Operations

HPF: C = A/B

Work: O(N) for N-element array A
Serial: O(N) time
Parallel: O(1) time on O(N) processors

SIMD heaven.

Data-centric parallelism: scan/reduce (parallel-prefix) operations

NESL: Sum(A)

Work: O(N) for N-element A
Serial: O(N) time
Parallel: O(log N) time on N processors
... plus log N network trips!

Not much help on low-dimensionality grid networks, but pretty good on tree networks.

If you can't do this efficiently, your machine sucks

Data-centric parallelism: scatter/gather

HPF scatter: ForAll(i = 0:1000:1) A[B[i]] = C[i]
HPF gather: ForAll(i = 0:1000:1) A[i] = C[B[i]]

Most general possible routing problem!
Essential for things like sorting

NESL gathers at the beginning of every conditional, and scatters at the end

How do you handle scatter collisions?

Data Placement Strategies

HPF: explicit alignment/realignment directives

NESL: dynamically compacts on every conditional

Java: hand-place remote objects

Cilk/pH: rely on shared memory implementation (especially caching!) to bring data where needed

Jade: copy or move objects to where they're needed

MIMD data placement strategies meet vectors

Cilk: threads on different processors writing adjacent vector elements (i.e. in same cache line) thrash the cache

Jade: object-level access specifications forbids parallel writing of a vector

Work placement strategies

HPF: uniform over processor array, except during conditionals

NESL: uniform at all times (heavy network load!)

pH: communal thread-queue in shared memory

Cilk: randomized work-stealing

Java/RMI: new threads run on same machine, remote call transfers to remote machine

Jade: central work-allocator feeds idle processors based on data locations

Forms of parallel communication

Message-passing:
- explicit points of communication, but
- very low-level
- difficult to program with

Shared memory:
- implicit communications via side-effects (including locks/semaphores, etc.)
- implicit form of broadcast
- nice way of sharing complex data structures
- easy to accidentally "communicate"

Aiming at your toes with shared memory

HPF: **"An assignment in a ForAll construct must not cause more than one value to be assigned to any array element."**

C/Cilk/Java: No lock? No luck.
Your program's behavior will depend on the vagaries of the thread scheduler!

pH: two writes to an M-structure will dynamically cause an explicit error; still scheduler-dependent!

Sequential semantics handcuffs (Jade)

Runtime checks of access specifications ensure:

- **two pieces of code which might conflict according to the specs are sequentialized**
- **a piece of code which tries to access something outside of specification raises an explicit error**
- **errors are deterministic: the same inputs will always generate the same errors!**

The bad news: threads can't cooperate back-and-forth.

Blowing your toes clean off: unverified assertions

HPF: "independent" -- do loop's innards are nominally independent

C: "noalias" -- these two variables never refer to the same object

cast from void * -- this object is really what I'm claiming it is

My opinion: if the compiler can't prove it at compile-time, the runtime better check it.

Opinionated summary: the state of the art

Languages fit into a couple of camps:

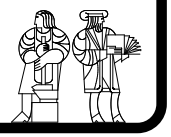
Massively parallel SIMD:
- **can use arbitrarily large machines**
- **require powerful communications networks**
- **limited domain of applicability**

Not-so-massively-parallel MIMD:
- **scalability issues**
  - **centralized schedulers**
  - **reliance on caching to colocate data and CPU**
  - **not so good at data-parallel ops**

Opinionated summary: MIMD challenges

Massively parallel MIMD programming languages and compilers need to provide:

- **good performance on SIMD-style problems**
  - **+ good data-layout strategy**
  - **+ good data/processing colocation strategy**
- **elegant side-effect management for inter-thread communications**
- **safety**
  - **+ incorrect results impossible**
  - **+ non-determinism only when explicitly invoked by the programmer**



Opinionated summary: performance

Hardware-level support for vector ops is good:
- hides scheduling overheads
- hides interpretive overheads
- hides latency overheads
- hides safety-check overheads
- etc.

One thread per datum will often be massively parallel inefficiency