YADA

David Gay, Susan Graham, Paul Hilfinger, Brian Kazian, Amir Kamil, Mayur Naik, Jimmy Su, and Katherine Yelick

June 5, 2008
Par Lab Research Overview

Easy to write correct programs that run efficiently on manycore

Personal Health | Image Retrieval | Hearing, Music | Speech | Parallel Browser

Design Patterns/Motifs

Composition & Coordination Language (C&CL)

C&CL Compiler/Interpreter

Parallel Libraries | Parallel Frameworks

Efficiency Languages

Sketching

Autotuners

Legacy Code

Schedulers

Communication & Synch. Primitives

Efficiency Language Compilers

Legacy OS

OS Libraries & Services

Hypervisor

Multicore/GPGPU

RAMP Manycore

Productivity Layer

Efficiency Layer

Diagnosing Power/Performance

Applications

Arch.

Static Verification

Type Systems

Directed Testing

Dynamic Checking

Debugging with Replay

Correctness
Yet Another Data Parallel Language

Main goals

- Focus on programmers productivity, not just efficiency
  - Provide reasonable per-core performance but good scalability
- Balance performance, productivity, and compiler complexity
- Irregular computation is a primary concern
- Target multicore hardware rather than large-scale parallel machines
Motivating Applications

- Heart blood-flow simulation
  - Developed by Peskin and McQueen at NYU
  - Applications
    - Understanding structural abnormalities
    - Evaluating artificial heart valves
    - Eventually, artificial hearts

- Par Lab health code (Tony Keaveny)

- Multimedia

Source: www.psc.org
Heart Model

- Composed of fibers in a fluid grid
- Includes atria, ventricles, valves, and some arteries
- The rest of the circulatory system is modeled by
  - sources: inflow
  - sinks: outflow
Heart Simulation Structure

4 phases in each timestep

1. Material activation & force calculation
2. Spread Force
3. Navier-Stokes Solver
4. Interpolate & move material

Material Points
Interaction
Fluid Lattice
2D Dirac Delta Function
Force Calculation Phase

- Calculates force on each fiber-particle
  - Force determined by positions of adjacent particles in the fiber according to Hooke’s law
- Fibers independent from each other
- Forces on different particles can be computed in parallel
  - Particle positions not updated, so no races

1. Material activation & force calculation
2. Spread Force
3. Navier-Stokes Solver
4. Interpolate & move material
Loop-Based Parallelism

- Explicitly parallel loops

```
Fiber fr = ..., fl = ...;
forall (x in f.particles.domain) {
    f[x].force = computeForce(fr[x].pos, fl[x].pos);
}
```

- Deterministic semantics: no races between iterations
  - Statically checked; warning and runtime checks when static verification fails
- Iterate on arrays, ranges, trees, graphs, user-defined types, and parallel iterators
Parallel Aggregate Operations

- Implicit aggregate operations
  \[ A = B + C \]
  - Equivalent to explicitly parallel loop
  \[
  \text{forall } (x \text{ in } A.\text{domain}) \\
  A[x] = B[x] + C[x];
  \]

- ZPL-style shifts and range restriction operators
  \[
  \text{Fiber } fr = f@right; \\
  \text{Fiber } fl = f@left;
  \]
Nested Parallelism

- Support nested parallelism
  \[
  \text{forall } (f \in \text{allFibers})
  \]
  \[
  \ldots
  \]
  \[
  \text{forall } (x \in f.\text{particles}.\text{domain})
  \]
  \[
  \ldots
  \]

- Two previous implementation strategies for nested parallelism
  - Flatten nesting: has only been applied to functional languages
  - Work stealing: has not been proven on data parallel languages
Force Calculation in YADA

forall (f in allFibers) {
    Fiber fr = f@right;
    Fiber fl = f@left;
    forall (x in f.particles.domain) {
        f[x].force = computeForce(fr[x].pos, fl[x].pos);
    }
}
Spread Force Phase

- Each particle spreads its force to its neighboring fluid cells
- A fluid cell may have multiple neighboring particles
- Updates to a fluid cell must be synchronized

2D Example
Accumulations and Reductions

- Programmer specifies accumulate/reduce operator by qualifying type of reduction target
  - Asserts indifference to order of application of operator
  - Research problem: prove order independence of user-defined functions

- Example: sum of elements of array \( A \)
  ```java
  int accumulate(+) sum = 0;
  forall (x in A) sum = sum + x;
  ```

- Multiple implementation strategies
  - Parallel tree reduction
  - Lock and operate
  - Transfer to owner and operate
Spread Force in YADA

double accumulate(+) [] force =
    new double[low:high];
forall (p in allParticles) {
    Point pos = [p.x, p.y, p.z];
    force[pos+north] += p.force;
    force[pos+east] += p.force;
    force[pos+south] += p.force;
    force[pos+west] += p.force;
}
Navier-Stokes Phase

- Incompressible fluid needs an elliptic solver
  - High communication demand
  - Information propagates across domain
- Uses FFT-based solver
  - Calls FFTW library to perform actual FFTs
- Need ability to call libraries written in other languages

1D FFTs

1. Material activation & force calculation
2. Spread Force
3. Navier-Stokes Solver
4. Interpolate & move material
Open Issues

- Base language
  - Previous data-parallel languages
  - Fortress/X10/Chapel
  - C family/Java/other sequential languages

- Precise feature set
  - Nested parallelism
  - ZPL-style shift operators
Moving Forward

- Implementation strategy
  - Initial prototype by Fall 2008 (serial? subset?)
  - Attempt to use existing serial and parallel libraries

- Performance goals
  - Good performance on simple data parallel code
  - Scalable performance on nested parallel code and other new features

- Early evaluation
  - Port heart code
  - Determine suitability for multimedia applications