Cerebral Blood Flow Simulation – Moving Towards a Parallel Implementation

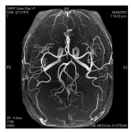
Meriem Ben-Salah, Chris Chaplin, Razvan Carbunescu, Farzana Ansari

Motivation

- Productivity layer programming language (e.g. Python) vs. efficiency layer programming language (e.g. C++):
  - 5x faster development and 3-10x fewer lines of code
  - BUT, 10x-100x less performance without exploring hardware model
- Scientists → Productivity layer Programmers
- ☹ Computationally expensive problems and lack of computer science knowledge -> Disaster!
- In need of a technique that permits:
  - high level knowledge of computer architecture abstraction
  - and simultaneously good performance
- SEJITS!
- Scientists/Productivity Layer Programmer input is needed

Cerebral Blood Flow Simulation

- Explore the SEJITS technique by means of a real application and define what the needs are?
- Personalized Medicine Application: Simulation of the blood flow in the main cerebral arteries (the Circle of Willis) based on patient data (CT scan, ultrasound) to save costs and save lives of stroke patients.



Numerics

- Numerical Scheme: Finite Volume Method in space and Forward Difference in time
- Complexity:
  - O(10^6) time steps, each time step is 10 microseconds long.
  - Maximal time: ~ 5 to 10 seconds
  - O(10^4) grid points, spatial grid size is 0.1 millimeter
  - Total length of vasculature: 1- 1.5 meter
  - O(160) Gigaflops
  - Time allotted: 30 seconds
  - Gigafloprate: 240 Gigaflops/second

Blood Flow Simulation

1. Input geometry of cerebral arteries from medical image
2. Assume an initial state of blood flow
3. Generate spatial discretization
4. March in time
   1. Calculate initial interface values
   2. Enforce boundary conditions
   3. Correct and evaluate final interface values
   4. Update the values at cell centers

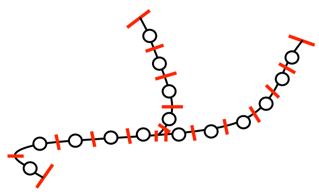
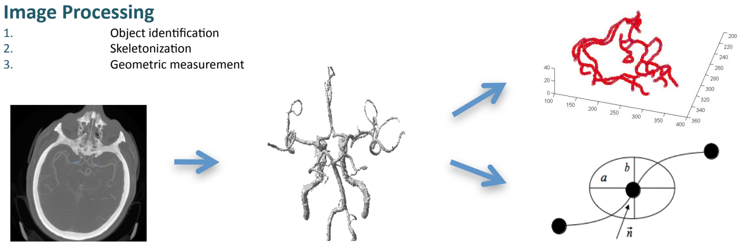


Image Processing

1. Object identification
2. Skeletonization
3. Geometric measurement



Current Implementation

- Blood flow simulation
  - Prototype in Matlab
  - Code tested on simple geometries, additional testing necessary for validation and verification
  - Incomplete Python version
- Image processing
  - Algorithm and code development underway for object identification and skeletonization; geometric measurement to come

Design Space

Structural Patterns:
- Pipe & Filter
- Agent & Repository
- Process Control
- Event-Based/Implicit Invocation
- Puppeteer
- Model-View-Controller
- Iterative Refinement
- Map-Reduce
- Layered Systems
- Arbitrary Static Task Graph

Computational Patterns:
- Graph-Algorithms
- Dynamic Programming
- Dense Linear Algebra
- Sparse Linear Algebra
- Unstructured Grids
- Structured Grids
- Graphical Methods
- Finite State Machines
- Backtrack-Branch-Bound
- N-Body-Methods
- Circuits
- Spectral Methods
- Monte Carlo

Parallel Algorithm Pattern:
- Task-Parallelism
- Divide and Conquer
- Data-Parallelism
- Pipeline
- Discrete-Event
- Geometric-Decomposition
- Speculation

Implementation Pattern:
- Program:
  - SPMD
  - Fork/Join
  - Loop-Parallel
  - Task-Queue
  - Data-Parallel
  - Actors
- Data:
  - Shared-Queue
  - Shared Map
  - Partitioned Graph
  - Distributed Array
  - Shared Data

Execution Pattern:
- MIMD
- SIMD
- Thread-Pool
- Task-Graph
- Transactions

Handled by Stovepipes

SEJITS--Selective Embedded Just In Time Specialization

- Libraries are written in Low Level Programming Languages and are too specific
- Stick with a High Level Programming Language e.g. Python
- Hidden to the productivity layer programmer but good to know:
  - Computational Patterns (good for reuse) are selected (if it is worth it) and specialized at runtime
  - A specialized function is written in a low level language targeted to the "parallel" hardware architecture (so called stovepipe technique)
  - Use of Autotuning techniques (e.g. in Pyski, autotuned Linear Algebra methods embedded in Python)
  - The low level code is compiled at run time
  - The low level code is dynamically linked using the high level language interpreter

SEJITS Implementation (CS 294 Class Project*)

- Majority of code has been implemented in Python using data types and linear algebra operations that are intrinsic to NumPy and SciPy
- Specializers have been developed for stencil operations and iterative solvers
- These specializers have been integrated with the Python code using ASP
- Introduced a structural "meta-specializer" in the form of a parallel executor that composes individual specializers into a single compilation unit
- Initial performance tests show an order of magnitude speed-up for the JIT-compiled version when compared to the pure-Python implementation

* Thanks to Ben Carpenter, James Ide, and Steven Liu for their work on this project

Future Work

Blood Flow Simulation

- Verify and validate the blood flow simulation using the prototype
- Finish the Python implementation of the model
- Compare run-time performance of code as applied to typical CT scans

Image Processing

- Refine the algorithm and prototype
- Convert code to Python and then exploit SEJITS
- Combine both codes to achieve an overall analysis time of ~60 seconds

Collaborators

Professors:		Undergraduates:	
	James Demmel		Sarah Tanaka
	Tony Keaveny		Walter Caliboso
	Panos Papadopoulos		Ben Carpenter
	Phil Colella		James Ide
	Max Wintermark		Steven Liu

Research supported by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227). Additional support comes from Par Lab affiliates National Instruments, NEC, Nokia, NVIDIA, Samsung, and Sun Microsystems.

Contacts: Meriem Ben-Salah bensalah@me.berkeley.edu, Chris Chaplin cmchap@berkeley.edu
Razvan Carbunescu carazvan@eecs.berkeley.edu