Optimization of the Poisson Operator
in Chombo

Razvan Carbunescu, Meriem Ben Salah
and
Andrew Gearhart

Research supported by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227)
Administrivia: The emu is watching you
Outline

- Introduction
  - Why the Poisson operator?
  - What is Chombo?
- Theoretical Background
- Targeted Architectures
- Implementation
- Challenges
Outline

- Results
  - Serial Implementation
  - pthreads
  - OpenMP
  - GPU (GTX280)

- Future Work
Introduction: Why the Poisson Operator?

- The Poisson operator is a key component of the Poisson equation
- A Poisson solution is the first step toward solving incompressible Navier-Stokes equations for fluid flow
- Parlab Health Application
  - Modeling blood flow through cerebral arteries

Figure: Flow Evolution startup after Poisson solve

Figure: Particle-In-Cell simulation from LBL using Chombo
**Introduction**: Why the Poisson Operator?

- The Poisson operator is a key component of the Poisson equation.

- A Poisson solution is the first step toward solving incompressible Navier-Stokes equations for fluid flow.

- Parlab Health Application
  - Modeling blood flow through cerebral arteries.

*Figure: Flow Evolution startup after Poisson solve*

*Figure: Particle-In-Cell simulation from LBL using Chombo*
Introduction: What is Chombo?

- Developed and distributed by the Applied Numerical Algorithms Group of Lawrence Berkeley National Lab

- A framework to implement finite difference methods for the solution of PDEs on block structured, adaptively refined grids.

- Chombo provides elliptic and time-dependent modules, as well as support for standardized self-describing file formats.

- Chombo is architecture and operating system independent.
Introduction: What is Chombo?

- For parallel platforms, Chombo provides a distributed memory implementation using the Message Passing Interface (MPI) library

- This begs the question:
Introduction: What is Chombo?

- For parallel platforms, Chombo provides a distributed memory implementation using the Message Passing Interface (MPI) library.

- This begs the question: Is a distributed memory implementation always the most efficient?
Introduction: What is Chombo?

- For parallel platforms, Chombo provides a distributed memory implementation using the Message Passing Interface (MPI) library.

- This begs the question: Is a distributed memory implementation always the most efficient?

- One of the major components of Chombo is a collection of Multigrid (MG) solvers for discretized elliptic problems, including Poisson's Equation.

- This portion of the Chombo suite has been modified to explore the above question.
Introduction: Strategy and Goals

- Determine whether we can improve Chombo performance through the use of locality and faster access time to on-chip cores
  - Via different models of parallel execution and specialized hardware (i.e., the GPU)

- Identify critical “crossover” points where one algorithm becomes more efficient (if they exist)

- Hopefully foster the creation of heterogeneous systems that automatically adapt execution to utilize distributed/shared memory or the GPU to enhance performance
Theoretical Background

- The Poisson operator, aka the Laplacian, is a second order elliptic differential operator and defined in an n-dimensional Cartesian space by:

\[ \Delta \Phi = \sum_{i=1}^{n} \frac{\partial^2 \Phi}{\partial x_i^2} \]

Given a function \( \Phi \) the Laplacian operator is given by

- The Poisson operator appears in the definition of the Helmholtz differential equation:

\[ \Phi + \beta \Delta \Phi = f \]

- The Helmholtz differential equation reduces to the Poisson equation:

\[ \text{if } \alpha = 0, \text{ then } \beta \Delta \Phi = f \]
Theoretical Background

- The definition of appropriate boundary conditions, Dirichlet or Neumann, allows for the solution of the Poisson problem.

- A numerical solution requires the discretization of the continuous Poisson’s equation, e.g. by the standard centered-difference approximation, as well as a discrete handling of the boundary conditions.

- The discrete Poisson operator, the focus of this project, is given by the following stencil:

\[
(\triangle^h \Phi_i) \approx \frac{1}{h^2} \sum_{d=0}^{D-1} (\Phi_{i+e^d} + \Phi_{i-e^d} - 2\Phi_i),
\]

where \( h \) is the grid size, \( i \) is the index of the cell-centered data of interest, \( D \) is the domain dimension, and \( e^d \) is the grid director.
Existing Implementations:

- Chombo’s implementation is currently tuned for distributed memory with the domain being decomposed into small bins (32^3 elements for 3D) and individual bin computations are allocated for execution via serial f77 codes.

- Because of the small size of bins there is a small amount computational intensity to use a threaded shared memory implementation or to hide the cost of GPU memory transfers.
Targeted Architectures

- **Our interest:**

  - Could Chombo be optimized for different models of parallel computation if the bin sizes were increased?

  - We would like to implement shared memory computation models utilizing:

    - **OpenMP**
    - **Pthreads**
      - lightweight threads may perform well with small bins
Targeted Architectures

- **Our interest:**
  
  - Another interesting opportunity for speedup is running the operator on the GPU
  
  - benefits must outweigh the data transfer cost
  
  - GPU execution offers a highly-parallel execution environment with proven performance for stencil codes
  
  - uncoalesced memory accesses can be problematic
implementation

- Chombo is implemented in C++ utilizing a complex set of templates and classes

- Bottom-level computation is performed in Fortran 77 via the Fortran/C interface

- Key components of the software package are grouped accordingly:

  - **BoxTools**: Calculations over unions of rectangles
  - **AMRTTools**: Communication between MG refinement levels
  - **EBTools**: Embedded boundary discretization
  - **AMRElliptic**: MG solvers on disc. elliptic and parabolic equations
  - **AMRTimeDependent**: Subcycling of time dependent computations
  - **ParticleTools**: Particle dynamics
Implementation

- Low-level C functions replace Fortran 77 kernels
  - Performance implications?

- Bad coding style:
  - “ghetto hack” or “feature development”?
  - abstract class hierarchy was bypassed to access data arrays directly
  - arrays are stored in Fortran column-major order, and then modified within C functions
    - problems with memory indexing
  - only have access to already-decomposed computational regions
    - limited to 2048 elements cubed
- **Methods:**
  
  - used GNU compiler suite: g++ and gfortran 4.2.0
  
  - Implemented Poisson operator with a C function instead of Fortran 77
  
  - A version of the C code utilizes the “__restrict__” type qualifier and “-fstrict-aliasing” to declare parameters as non-aliased
Results: Serial Implementations

Problem Size vs. Serial code runtime (applyOp)
Results: Pthreads

- **Methods:**

  - Utilized the standard pthread library to implement a parallel code that runs on-chip without the overhead of MPI
  
  - Threaded code was run on NERSC's Cray XT4 ("Franklin")
    - Quad-core, 64-bit AMD Opteron nodes

  - Codes were run with 4 threads to explore node-local performance
Results: Pthreads

Problem Size vs. Speedup over C serial (applyOp)
Results: OpenMP and GPU

- Methods:
  - OpenMP
    - The OpenMP implementation is implemented via code directives that indicate parallel sections of code
    - This promises access to an abstract and powerful way to parallelize codes
  - GPU
    - Due to convergence problems in single-precision, the double-precision nVidia GTX280 was the focus of experimentation
    - Stencil code was written in using NVidia's Cuda extensions to the C programming language
Results: OpenMP and GPU

- Data collection pending:
  - Compilation errors for OpenMP code
  - Indexing for the Cuda version of the solve is currently in error
  - Currently, the Cuda stencil is a very naïve implementation and does not optimize using blocking for registers and shared memory
Future Work

- more complicated memory optimizations
  - circular queues
  - time skewing

- better GPU memory coalescing
  - blocking
  - padding

- using the GPU's other memory
  - constant
  - texture

- interpolation via texture cache hardware
Thanks to all our colleagues at LBL and at UC Berkeley for their gracious help in the development of this project.
Administrivia: We got the emu