

CMSC 714

Lecture 7

PETSc and Cactus

Alan Sussman

Notes

- MPI project due yesterday
 - Questions?
- OpenMP project posted today – due Oct. 11

PETSc

- Portable, Extensible Toolkit for Scientific Computation
- Library to encapsulate commonly used functions and data structures for numerically solving partial differential equations
- Targeted at message passing for scalability, but hides it (mostly) from application
- Uses object-oriented programming techniques
 - Data encapsulation
 - Polymorphism
 - Inheritance
 - but implemented in C, so no compiler support
- Essentially SPMD style programming, but w/o explicit message passing

6 guiding principles

- For performance
 - overlap communication and computation
 - determine details of repeated communication patterns, and optimize message passing across multiple calls (inspector/executor model)
 - allow user to decide when communication occurs (if needed)
 - allow user to aggregate data for later communication
- For ease of use
 - allow user to work on distributed objects (arrays) without knowing which processor owns which data elements
 - manage communication at higher levels, on objects, instead of directly using message passing

Distributed Objects

- Low level data structures
 - Vectors
 - Matrices
 - Index Sets
- Low level algorithms
 - Create and assemble a vector or matrix – vector scatter/gather, sparse matrix examples in paper
- Higher level algorithms
 - PDE solvers
 - Linear and non-linear equation solvers
 - Time steppers
 - Preconditioners
- All functions take an MPI_Comm as an argument

Six Guiding Principles (again)

- Managing communication within higher level data structures and algorithms
 - MPI calls generated to perform communication needed to perform higher level ops on distributed objects
 - Implication is no optimizations across calls
- Overlap communication and computation
 - Separate start and end of complex operations, so other computations can go on in between, like MPI non-blocking operations
- Precomputing communication patterns
 - Generate a pattern of sends/receives for an operation on a distributed object (which may need communication), then reuse the pattern for subsequent data movement operations
 - Often called inspector/executor model

Guiding Principles (cont.)

- Programmer management of communication
 - User can explicitly start and end communication via specific PETSc calls
 - Often to enable overlap of communication with computation
- Work on distributed objects, not on individual data elements
 - Avoids programmer having to move data between application data structures and library data structures
 - Can build PETSc data structures from any process, with data for any process (not just local to a process)
 - This is what is meant by “assembly”
- Aggregate data for communication
 - To minimize number of messages
 - Communication cost proportional to number of messages, plus per byte cost

Cactus

- Application framework, mainly targeted at astrophysics and relativity apps
 - And other multidisciplinary apps
- Hides data distribution and other performance related programming issues from application logic
 - Data distribution, message passing, parallel I/O, scheduling, etc.
- Also targets computational Grids
 - Distributed sets of HPC resources
- Based on earlier frameworks
 - DAGH, GrACE for parallelizing solution of complex sets of differential equations (Einstein's equations for relativity)
- Core is called the “flesh”, user-defined modules are called “thorns”

Design criteria

- **Run on a wide variety of machines**
 - From desktop to large scale parallel
 - So need a flexible, modular build system – need to auto-detect system properties to minimize user configuration – based on autoconf/automake
- **Should be easy to add new modules**
 - Need separate name spaces for data for each module (thorn) so they can co-exist
 - Functionally equivalent modules should be interchangeable
- **Transparent support for parallelism**
 - Abstractions for distributed arrays, data parallelism, data decomposition, synchronization, etc.
 - And should be architecture independent
- **Input and output modules also thorns, so can be used by other thorns transparently**
 - Including parallel I/O, support for different file formats
- **Support legacy code by making them easy to wrap as thorns**

Cactus program structure

- **Flesh – the core**
 - Provides main program to parse parameters and call thorns
 - Mostly a means to move things around in memory
- **Thorn**
 - Contains all user code
 - Communicates with other thorns via calls to flesh API, and sometimes calls to custom APIs of other thorns
 - Can be written in C, C++, or Fortran (77 or 90)
- **Connections from thorn to flesh (or other thorns) through configuration file that is parsed at compile time**
 - Glue code generated to encapsulate thorn
- **Configuration is a build of flesh and set of thorns for a given architecture with config options**

More on thorns

- **Grid variables – externally visible to flesh and other thorns, so are related to overall computation**
 - Grid Scalars – single numbers (per process)
 - Grid Functions – distributed arrays with size set by overall problem size (the grid size for the discretized equations)
 - Grid Arrays – distributed arrays of any size
- **Thorn provides specification files written in CCL (Cactus Configuration Language)**
 - Say what functions the thorn implements (and their interfaces)
 - Variables (data) that need to be supplied (from other thorns, via the flesh)
 - Parameters thorn uses
 - What routines must be called (and in what order)

Scheduling thorns

- **Thorn routines can be scheduled to run via a rule specification**
 - A routine can be scheduled before or after other routines from the same or other thorns
 - And while some condition is true
 - e.g., an overall computation termination condition
- **Routines registered with scheduler, and the overall set of specs generates a DAG, which can then be executed multiple times (in topologically sorted order)**
 - Scheduler either part of flesh, or a separate thorn (not clear from paper)

Driver thorns

- Responsible for memory management for grid variables, and for parallel operations
 - As asked by the scheduler
 - So can distribute arrays for parallel execution (typically message passing SPMD style, but could be shared memory too)
- Three parallelization/synchronization operations
 - Ghost-zone updates between sub-domains (across boundaries of a distributed array)
 - Generalized reductions (combine values contributed by different processes)
 - Generalized interpolation (to perform more complex transformations on data at grid coordinates)
- Thorn routines can request synchronization of grid variables on exit
- Four known driver thorns
 - One grid, non-parallel (**SimpleDriver**)
 - One grid, parallel (**PUGH**) – seems to be the one most used
 - Parallel, fixed mesh refinement (**Carpet**) - multigrid
 - Parallel, adaptive mesh refinement (**PAGH**)