Kripke - User Manual v1.0

Author: Adam J. Kunen, LLNL

June 4, 2014

Table of Contents

Overview

Introduction

Kripke is a simple, scalable, 3D Sn deterministic particle transport code. Its primary purpose is to research how data layout, programming paradigms and architectures effect the implementation and performance of Sn transport. A main goal of Kripke is investigating how different data-layouts effect instruction, thread and task level parallelism, and what the implications are on overall solver performance.

Requirements

Kripkie supports storage of angular fluxes (Psi) using all six striding orders (or "nestings") of Directions (D), Groups (G), and Zones (Z), and provides computational kernels specifically written for each of these nestings. Most Sn transport codes are designed around one of these nestings, which is an inflexibility that leads to software engineering compromises when porting to new architectures and programming paradigms. Early research has found that the problem dimensions and the scaling (number of threads and MPI tasks), can make a profound difference in the performance of each of these nestings. To our knowledge this is a capability unique to Kripke, and should provide key insight into how data-layout effects Sn performance. An asynchronous MPI-based parallel sweep algorithm is provided, which employs the concepts of Group Sets (GS) and Direction Sets (DS), borrowed from the Texas A&M code PDT.

As we explore new architectures and programming paradigms with Kripke, we will be able to incorporate these findings and ideas into our larger codes. The main advantages of using Kripke for this exploration is that it's light-weight (ie. easily refactored and modified), and it gets us closer to the real question we want answered: "What is the best way to layout and implement an Sn code on a given architecture+programming-model?" instead of the more commonly asked question "What is the best way to map my existing Sn code to a given architecture+programming-model?".

Analysis

A major challenge of achieving high-performance in an Sn transport (or any physics) code is choosing a data-layout and a parallel decomposition

that lends itself to the targeted architecture. Often the data-layout determines the most efficient nesting of loops in computational kernels, which then determines how well your inner-most-loop SIMDizes, how you add threading (pthreads, OpenMP, etc.), and the efficiency and design of your parallel algorithms. Therefore, each nesting produces different loop nesting orders, which provides substantially different performance characteristics. We want to explore how easily and efficiently these different nestings map to different architectures. In particular, we are interested in how we can achieve good parallel efficiency while also achieving efficient use of node resources (such as SIMD units, memory systems, and accelerators).

Parallel sweep algorithms can be explored with Kripke in multiple ways. The core MPI algorithm could be modified or rewritten to explore other approaches, domain overloading, or alternate programming models (such as Charm++). The effect of load-imbalance is an understudied aspect of Sn transport sweeps, and could easily be studied with Kripke by artificially adding more work (ie unknowns) to a subset of MPI tasks.

Block-AMR could easily be added to Kripke, which would be a useful way to explore the cost-benefit analysis of adding AMR to an Sn code, and would be a way to further study load imbalances and AMR effects on sweeps.

The coupling of on-node sweep kernel, the parallel sweep algorithm, and the choices of decomposing the problem phase space into GS's and DS's impact the performance of the overall sweep. The tradeoff between large and small "units of work" can be studied. Larger "units of work" provide more opportunity for on-node parallelism, while creating larger messages, less "sends", and longer sweep stage times. Smaller "units of work" allow for faster on-node kernel times and faster pipe-fill times, but at the cost of more messages and total latency costs. We can also study trading MPI tasks for threads, and the effects this has on our programming models and cache efficiency.

A simple timer infrastructure is provided that measure each compute kernels total time.

Design

Through object-oriented C++ design, Kripke represents each nesting as an implementation of a Kernel interface class. This interface provides an abstraction layer between the rest of the code and the details of allocating storage, setup, implementation of kernels and data comparisons. This approach allows us to choose a nesting at run time. Furthermore, multiple nestings can be run on the same data and their results compared, which allows for correctness checking when making modifications to kernels. The numerics of these kernels are identical in complexity and accuracy to existing production codes.

Kripke provides spatial decomposition across MPI tasks, where each MPI task contains all of the groups and directions, but only for a subset of the zones. GS's are swept sequentially and DS's are pipe-lined through the MPI sweep. This makes Psi a 5-dimensional vector as Psi[GS][DS][G][D][Z], where the nesting of GS and DS are fixed, and GDZ are interchangeable as noted above. The MPI sweep is asynchronously scheduled, provides overlapped communication and compute (using isend/irecv) and starts the sweep at all corners of the spatial domain. Since this represents less than 400 LOC, implementing new approaches to this parallel sweep algorithm (such as using Charm++) would be easy and would be interesting avenues of research.

The "unit of work" done at each stage in the parallel algorithm is a sweep over zones, performed on a set of directions, groups and zones. This is implemented with Diamond Difference spatial discretization. Between full sweels (all GS and DS), we model the source terms by computing the spherical harmonic moments of the the angular flux with the LTimes kernel ("discrete to moments"), and then back to discrete space with the LPlusTimes kernel ("moments to discrete"). In order to leave this as a general purpose Sn transport code we provide no other computation of source terms. Through profilile real Sn codes, we believe that capturing the parallel-sweep, the "on-core" sweep kernel and the LTime and LPlusTimes operators faithfully represents the performance characteristics of a real Sn transport code.

Kripke has been implemented with C++ and MPI. No threading code or optimizations (like "restrict" keywords) have been added. The intent is to provide a "plain-vanilla" implementation in order to make the base implementations as easy to understand as possible. We also didn't want to inadvertently tie Kripke to some architectural or language features.

Kripke was not designed as a real transport solver, rather it is designed to represent Sn transport performance and explore implementation details. It doesn't have a notion of materials on a mesh or any kind of material information. It doesn't have a way to set initial conditions or extract solution information. There is no real iterative solver or convergence criteria. There is no way to provide cross sections or opacities. The transformation matrices LTimes an LPlusTimes are not computed from a quadrature set, rather are set to zeros.

Implementation

The source code is divided into individual ".cpp" files, roughly one per class or function.

The file "src/Kripke/Kernel.h" defined the Kernel interface and defined a factory function for creating objects. The implemented kernels are in "src/Kripke/Kernel/*.cpp". This is where most of the optimizations and programming model changes will occur.

The file "src/Kripke/Sweep_Solver.cpp" provides the MPI parallel sweep algorithm.

The rest of the files define the data structures that support these kernels and algorithms.

For a given problem, a single User_Data object is created, which contains everything: basic global parameters, quadrature set, MPI communicator object (for Sweep_Solver), and a Grid_Data object.

The Grid_Data object contains L and L+ matrices, variables in moments space (phi), and a set of Group_Dir_Set objects for each <GS,DS> pair.

It is the Group_Dir_Set objects which store the angular fluxes(psi) and right hand side (rhs). In our implementation, we always ensure that all directions contained in a single Group_Dir_Set object have the same sweeping order.

Physical models

Kripke solves a simplification of the Discrete Ordinance and Diamond Difference discretized steady-state linear Boltzmann equation:

$$(\Omega \cdot \nabla + \sigma)\, \Psi = L^{+} S L \Psi$$

where (in this case) S is the identity matrix, and the 3D Diamond-Difference is used for inverting the "streaming and collision" operator (omega dot grad + sigma). The kernels LTimes and LPlusTimes provide the action of the matrices L and L+, respectively.

One iteration of the MPI algorithm combined with the on-node sweep kernel solves one iteration of:

$$\Psi_{i+1} = (\Omega \cdot \nabla + \sigma)^{-1} (L^{+} S L \Psi_i)$$

Inputs and outputs

The Kripke build system produces a single executable "kripke". All of the parameters are supplied via command line options.

The number of GS and G (groups per groupset) are specified with "–grp <GS>:<G>". For example "–grp 16:2" specifies 16 groupsets and 2 groups per set, for a total of 32 groups.

Directions are specified *almost* the same way, with "–dir <DS>:<D>". However DS represents the number of *direction sets PER octant*. Therefore "–dir 2:4" represents 2 direction sets per octant, 4 directions per set, and a total of 64 directions.

Nestings are selected with the "–nest DGZ" option, where any of the six DGZ, DZG, GDZ, GZD, ZDG, ZGD are allowed.

Number of Legendre moments ("–legendre") are selectable from 0 up to specify the number of moments in L and L+.

Number of zones (for the entire domain) are selected with "–zones X,Y,Z", number of MPI tasks are selected with "–procs X,Y,Z".

Number of iterations can be selected with "–niter N".

Search spaces can be specified by supplying *lists* to three arguments "–dir", "–grp", and "–nest". The product of these sets defines an entire search space, all combinations of which will be run.

For example: "kripke --grp 1:4,2:2,4:1 --dir 16:1 --nest GDZ,ZDG" will run 6 different search points.

We defined four standard test problems which we feel exercise different scaling and performance aspects of any Sn transport solver.

Name	Directions (per Octant)	Groups	Scattering Order	Zones/Core	sizeof(psi) MBytes	sizeof(phi) MBytes
KP0	96 (12) ~S8	64	P4	1728 (12x12x12)	81.0	13.5
KP1	256 (32) ~S12	64	P4	1728 (12x12x12)	216.0	13.5
KP2	96 (12)	256	P4	1728 (12x12x12)	324.0	54.0
KP3	96 (12)	64	P9	1728 (12x12x12)	81.0	68.344

Examples of how to invoke *kripke* with these parameters is provided as example scripts with the source code.

Kripke outputs timing information for each search point it runs. For large search spaces an output file can be generated which allows for easy post-processing and plotting of results.

Testing

Testing can be performed with the "–test" command line option. This executes the problem specified by command line argument in a test harness mode. In this mode, each search point is compared with the results of the DGZ nesting. Each kernel is run through its own setup, test, and

comparison. Random values are supplied for the initial conditions, LPlus matrix, and LPlusTimes matrix. Then all variables (Psi, Phi, etc.) are compared, and any differences are printed to stdout.

This is very useful for determining correctness of compute kernels after each modification.

Future Plans/Retirement

Adding more physics, such as a scattering kernel, would help bring the

Domain overloading would allow us to explore more complex load-balancing techniques and the effect on parallel efficiency.

Block AMR.

1D and 2D kernels could be added relatively easily.

More FLOP intensive spatial discretizations such as DFEM's.

Retirement of this Mini-App should be considered when it is no longer a representative of state-of-the-art transport codes, or when it becomes too cumbersome to adapt to advanced architectures. Also, at the point of retirement it should be clear how to design its successor.

Using Kripke

Building Kripke

In the root path of the tarball, run the setup script to generate a build-space and initialize CMake. You need to have an MPI compiler, so specify that through the standard environment variables (only CXX actually gets used during the build):

```
>> CXX=mpig++ ./setup.py

SYS_TYPE:  Linux
PROJECT:   kripke
BUILD DIR: ./kripke-Linux
-- The C compiler identification is GNU 4.4.7
-- The CXX compiler identification is GNU 4.4.7
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working CXX compiler: /usr/local/bin/mpig++
-- Check for working CXX compiler: /usr/local/bin/mpig++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Configuring done
-- Generating done
-- Build files have been written to: XXX/kripke/kripke-Linux
```

The setup script has made the subdirectory *./kripke-Linux*. You can now build Kripke:

```
>> cd kripke-Linux
>> make

Scanning dependencies of target kripkelib
[  6%] Building CXX object src/CMakeFiles/kripkelib.dir/Kripke/Comm.cpp.o
[ 12%] Building CXX object src/CMakeFiles/kripkelib.dir/Kripke/Directions.cpp.o
[ 18%] Building CXX object src/CMakeFiles/kripkelib.dir/Kripke/Driver.cpp.o
[ 25%] Building CXX object src/CMakeFiles/kripkelib.dir/Kripke/Grid.cpp.o
[ 31%] Building CXX object src/CMakeFiles/kripkelib.dir/Kripke/Kernel.cpp.o
[ 37%] Building CXX object src/CMakeFiles/kripkelib.dir/Kripke/Sweep_Solver.cpp.o
[ 43%] Building CXX object src/CMakeFiles/kripkelib.dir/Kripke/Timing.cpp.o
[ 50%] Building CXX object src/CMakeFiles/kripkelib.dir/Kripke/User_Data.cpp.o
[ 56%] Building CXX object
src/CMakeFiles/kripkelib.dir/Kripke/Kernel/Kernel_3d_GDZ.cpp.o
[ 62%] Building CXX object
src/CMakeFiles/kripkelib.dir/Kripke/Kernel/Kernel_3d_DGZ.cpp.o
[ 68%] Building CXX object
src/CMakeFiles/kripkelib.dir/Kripke/Kernel/Kernel_3d_ZDG.cpp.o
[ 75%] Building CXX object
src/CMakeFiles/kripkelib.dir/Kripke/Kernel/Kernel_3d_DZG.cpp.o
[ 81%] Building CXX object
src/CMakeFiles/kripkelib.dir/Kripke/Kernel/Kernel_3d_ZGD.cpp.o
[ 87%] Building CXX object
src/CMakeFiles/kripkelib.dir/Kripke/Kernel/Kernel_3d_GZD.cpp.o
Linking CXX static library libkripkelib.a
[ 87%] Built target kripkelib
Scanning dependencies of target kripketools
[ 93%] Building CXX object src/tools/CMakeFiles/kripketools.dir/testKernels.cpp.o
Linking CXX static library libkripketools.a
[ 93%] Built target kripketools
Scanning dependencies of target kripke
[100%] Building CXX object src/tools/CMakeFiles/kripke.dir/kripke.cpp.o
Linking CXX executable kripke
[100%] Built target kripke
```

That's it! No you can run Kripke.

Command Line Options

All input parameters are supplied to Kripke by command line arguments. No input files are used. The use of these arguments is described in a previous section. Following is a description of each of these command line arguments.

Argument	Default	Description
--dir **D**:**d**[,**D**:**d**[...]]	--dir 1:1	List of Direction Set and Directions-per-Set pairs. **D** is the number of direction sets *per octant*, and **d** is the number of directions per set. Therefore, the default total number of directions is 8. Lists of direction parameters may also be supplied, for example: *(no spaces in list)* --dir 1:4,2:2,4:1

--grp **G**:**g**[,**G**:**g**[...]]	--grp 1:1	List of Group Set and Groups-per-Set pairs. **G** is the number of group sets, and **g** is the number of groups per set. Therefore, the default total number of groups is 1. Lists of group parameters may also be supplied, similarly to --dir.
--legendre **L**	--legendre 4	Scattering Legendre expansion order. This is used to determine the number of moments for the L and L+ matrices. Valid values are 0,1,2,... where 0 would represent isotropic scattering.
--nest **N**[,**N**[...]]	--nest DGZ,DZG,GDZ,GZD,ZDG,ZGD	List of data nestings. These are the inner-most 3 data nestings used for Psi and Phi. Kernels and storage specific to the nesting are used for the solve.
--niter **NITER**	--niter 10	Number of solver iterations to run. A number large enough should be chosen to drive down statistical noise in the timing results.
--out **OUTFILE**	*none*	Optional output file. This file will contain the timing results for each point in a machine/human readable format. This output format is used by the post-processing scripts provided.
--procs **NX**,**NY**,**NZ**	--procs 1,1,1	MPI task spatial decomposition. This is the number of MPI tasks in X, Y, and Z. The total number of MPI tasks in the run must equal X*Y*Z.
--test	*off*	This flag turns off the normal solver mode, and turns on the kernel testing suite. It runs each point specified by the command line arguments, but compares the results of each kernel to DGZ.
--zones **X**,**Y**,**Z**	--zones 12,12,12	Total number of spatial zones in X,Y, and Z. The number of zones per MPI task is roughly X/NX by Y/NY by Z/NZ. Therefore, with "–zones 12,12,12 --procs 2,2,2", each of the 8 MPI tasks would have a 6x6x6 spatial grid.

Examples

These examples assume you are have setup and built Kripke, and that your current working directory is your build directory.

Running Single Point from KP0

```
>> ./src/tools/kripke --zones 12,12,12 --dirs

------------------------------------------------------------
------------------- KRIPKE VERSION 1.0 ---------------------
------------------------------------------------------------
Number of MPI tasks:   1
Output File:
Processors:            1 x 1 x 1
Zones:                 12 x 12 x 12
Legendre Order:        4
Number iterations:     10
GroupSet/Groups:       8:8
DirSets/Directions:    12:1
Nestings:              DGZ
Search space size:     1 points
Running point 1/1: D:d=12:1, G:g=8:8, Nest=DGZ
nestid=0 nest=DGZ D=12  d=1   dirs=96 G=8   g=8   grps=64 Solve=2.2660   Sweep=0.4426
LTimes=0.8974   LPlusTimes=0.9260

>>
```

Running Multiple Points from KP0 with output file

```
>> ./src/tools/kripke --dir 12:1 --grp 1:64,8:8,64:1 --legendre 4 --zones 12,12,12
--nest DGZ,DZG --out test.out

------------------------------------------------------------
------------------- KRIPKE VERSION 1.0 -------------------
------------------------------------------------------------
Number of MPI tasks:   1
Output File:           test.out
Processors:            1 x 1 x 1
Zones:                 12 x 12 x 12
Legendre Order:        4
Number iterations:     10
GroupSet/Groups:       1:64, 8:8, 64:1
DirSets/Directions:    12:1
Nestings:              DGZ, DZG
Search space size:     6 points
Running point 1/6: D:d=12:1, G:g=1:64, Nest=DGZ
nestid=0 nest=DGZ D=12  d=1   dirs=96 G=1   g=64  grps=64 Solve=2.3759   Sweep=0.4345
LTimes=0.9638   LPlusTimes=0.9777
Running point 2/6: D:d=12:1, G:g=1:64, Nest=DZG
nestid=1 nest=DZG D=12  d=1   dirs=96 G=1   g=64  grps=64 Solve=2.9607   Sweep=0.2378
LTimes=1.4598   LPlusTimes=1.2631
Running point 3/6: D:d=12:1, G:g=1:64, Nest=DGZ
nestid=0 nest=DGZ D=12  d=1   dirs=96 G=8   g=8   grps=64 Solve=2.2436   Sweep=0.4392
LTimes=0.8887   LPlusTimes=0.9157
Running point 4/6: D:d=12:1, G:g=1:64, Nest=DZG
nestid=1 nest=DZG D=12  d=1   dirs=96 G=8   g=8   grps=64 Solve=8.8822   Sweep=0.4659
LTimes=4.3227   LPlusTimes=4.0936
Running point 5/6: D:d=12:1, G:g=1:64, Nest=DGZ
nestid=0 nest=DGZ D=12  d=1   dirs=96 G=64  g=1   grps=64 Solve=2.2853   Sweep=0.4555
LTimes=0.9035   LPlusTimes=0.9263
Running point 6/6: D:d=12:1, G:g=1:64, Nest=DZG
nestid=1 nest=DZG D=12  d=1   dirs=96 G=64  g=1   grps=64 Solve=37.6436  Sweep=2.2048
LTimes=17.8140  LPlusTimes=17.6247

>> cat test.out
nestid=0 nest=DGZ D=12  d=1   dirs=96 G=1   g=64  grps=64 Solve=2.3759   Sweep=0.4345
LTimes=0.9638   LPlusTimes=0.9777
nestid=1 nest=DZG D=12  d=1   dirs=96 G=1   g=64  grps=64 Solve=2.9607   Sweep=0.2378
LTimes=1.4598   LPlusTimes=1.2631
nestid=0 nest=DGZ D=12  d=1   dirs=96 G=8   g=8   grps=64 Solve=2.2436   Sweep=0.4392
LTimes=0.8887   LPlusTimes=0.9157
nestid=1 nest=DZG D=12  d=1   dirs=96 G=8   g=8   grps=64 Solve=8.8822   Sweep=0.4659
LTimes=4.3227   LPlusTimes=4.0936
nestid=0 nest=DGZ D=12  d=1   dirs=96 G=64  g=1   grps=64 Solve=2.2853   Sweep=0.4555
LTimes=0.9035   LPlusTimes=0.9263
nestid=1 nest=DZG D=12  d=1   dirs=96 G=64  g=1   grps=64 Solve=37.6436  Sweep=2.2048
LTimes=17.8140  LPlusTimes=17.6247

>>
```

Testing

```
>> ./src/tools/kripke --dir 12:1 --grp 8:8 --legendre 4 --zones 12,12,12 --nest DZG
--test

------------------------------------------------------------
------------------- KRIPKE VERSION 1.0 ---------------------
------------------------------------------------------------
Number of MPI tasks:   1
Output File:
Processors:            1 x 1 x 1
Zones:                 12 x 12 x 12
Legendre Order:        4
Number iterations:     10
GroupSet/Groups:       8:8
DirSets/Directions:    12:1
Nestings:              DZG
Search space size:     1 points
Running point 1/1: D:d=12:1, G:g=8:8, Nest=DZG
  Comparing GDZ to DZG for kernel LTimes
    -- allocating
    -- randomizing data
    -- running kernels
    -- comparing results
    -- OK

  Comparing GDZ to DZG for kernel LPlusTimes
    -- allocating
    -- randomizing data
    -- running kernels
    -- comparing results
    -- OK

  Comparing GDZ to DZG for kernel Sweep
    -- allocating
    -- randomizing data
    -- running kernels
    -- comparing results
    -- OK

>>
```

Contact Information

Adam J. Kunen – kunen1@llnl.gov

Peter N. Brown – brown42@llnl.gov

Teresa S. Bailey – bailey42@llnl.gov