Nim for LQCD

Tuning force-gradient integrators for 8 flavor nHyp HMC

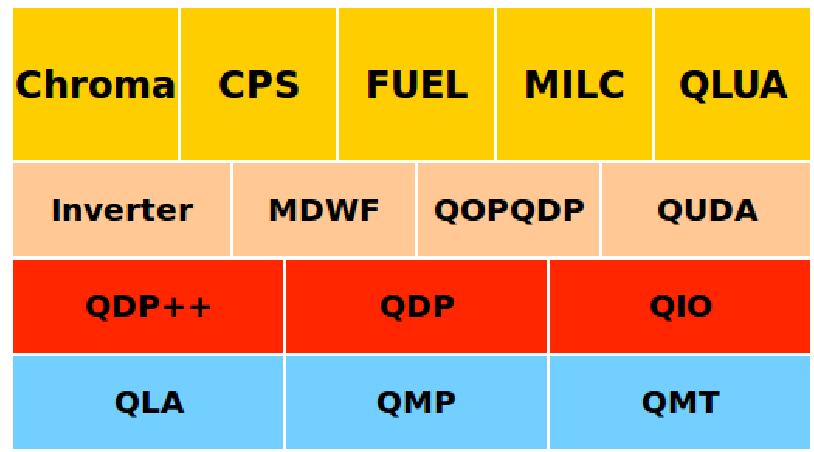
James C. Osborn & Xiao-Yong Jin

Argonne Leadership Computing Facility

August 1
QCDNA 2016
Edinburgh, UK

Evolution of USQCD SciDAC "C" software

- Shared base (in C): QMP, QIO
- C/C++ data parallel: QDP+QLA, QDP++
- QOPQDP: solvers, forces, etc. built on QDP
- Lua application scripting layers on QDP/QOPQDP: QLUA, FUEL
- Lua scripting provides
  - Ease of use
  - Rapid development & testing
  - Speed of C underneath



- QLA/QDP
  - Array of structures
  - Originally no threading (now has OpenMP)
  - Needs modern update

Evolution of USQCD SciDAC C/Lua software

- Started new framework to experiment with threading and vectorization (QLL)
- Hand written + Lua generated C code
- Well tuned staggered + Naik CG gets 23% of peak on BG/Q
- Started looking for high-level language
  - Transform natural expressions into well optimized code
  - Have ability to perform optimizations across multiple expressions (i.e. loop fusion)
- Discovered (nearly*) perfect language for the job: Nim

* “not perfect yet”

Nim (nim-lang.org)

- Modern language started in 2008
- Designed to be “efficient, expressive, and elegant”
- Borrows heavily from: Modula 3, Delphi, Ada, C++, Python, Lisp, Oberon
- Statically typed, but has extensive type-inference, so feels like dynamically-typed scripting language
- Efficient garbage collection (optional)
- Extensive meta-programming support (nearly full language available at compile time)
- Still young for language
  - Current version 0.14.2
  - Strong desire to work towards 1.0 (backward stability)
  - Small, but growing community (users and developers)

Nim

- Nim compiles to C/C++ (also JS, PHP): “one level up” from C/C++

  C++ → (clang) → IR → (LLVM) → ASM → (as) → obj → (ld) → binary

  Nim

- C/C++ backend provides
  - Portability
  - Easy integration with C/C++ libraries, intrinsics (simd), pragmas (OpenMP, OpenACC), OpenCL, CUDA(?)
- integrated build system tracks dependencies, compiles and links:
  - no Makefile necessary: copy main program, modify, compile

    nim c myProject1.nim
    nim c myProject2.nim
    …

Generic and meta-programming features

C++	Nim
preprocessor macros	templates: inline code substitutions also allows overloading, completely hygenic (if desired)
templates	generics: applies to type definitions, procedures, templates and macros also allows typeclasses, concepts
???	macros: similar to lisp: syntax tree of arguments passed to macro at compile time to allow arbitrary manipulation

Simple macro example

- Transform loops
- Standard for loop:

```
for i in 0..2:
  foo(i)
```

- macro:

```
macro forStatic(index:untyped; slice:Slice[int]; body:untyped):stmt = ...

forStatic i, 0..2:
  foo(i)
```

→

```
foo(0)
foo(1)
foo(2)
```

Macros for low level optimization

- optimize:
  var t: array[3, tuple[re: vector4double, im: vector4double]]
  …
  t[0].re = ...
  t[0].im = ...
  ...

  →

  var t0re: vector4double
  var t0im: vector4double
  …
  foo(t0re)
  foo(t0im)
  …

Tensor operations (Xiao-Yong Jin)

- General tensor support in development:

```
tensorOps:
  v2 = 0
  v2 += v1 + 0.1
  v3 += m1 * v2
```

`(above code block transforms to the pseudocode)`

```
for j in 0..2:
  v2[j] = 0
  v2[j] += v1[j] + 0.1
  for k in 0..2:
    v3[k] += m1[k,j] * v2[j]
```

- `Can also use Einstein notation (autosummation):`

```
v1[a] = p[mu,mu,a,b] * v2[b]
```

New lattice framework in Nim: QEX (Quantum EXpressions)

- Using layout/communications framework from QLL
  (will eventually convert to Nim, not urgent: Nim works great with C)
- Working example of staggered solver (plain & Naik) & simple meson analysis
- Plan to work on link smearings + HMC soon
- Linear algebra undergoing reorganization
  - Optimizations and tensor support
- Once more code is running, will shift focus to improving high-level interface
- Code available on github
  https://github.com/jcosborn/qex

QEX: QCD (or Quantum) Expressions

```
import qex
import qcdTypes

qexInit()
var lat = [4,4,4,4]
var lo = newLayout(lat)
var v1 = lo.ColorVector()
var v2 = lo.ColorVector()
var m1 = lo.ColorMatrix()
threads:
  m1 := 1
  v1 := 2
  v2 := m1 * v1
  shift(v1, dir=3, len=1, v2)  # len=+1: from forward
  single:
    if myRank==0:
      echo v2[0][0]  # vector “site” 0, color 0
qexFinalize()
```

QEX/Nim examples

- threads: implementation

```
template threads*(body:untyped):untyped =
  let tidOld = tid
  let nidOld = nid
  proc tproc =
    {.emit:"#pragma omp parallel".}
    block:
      setupForeignThreadGc()
      tid = ompGetThreadNum()
      nid = ompGetNumThreads()
      body
  tproc()
  tid = tidOld
  nid = nidOld
```

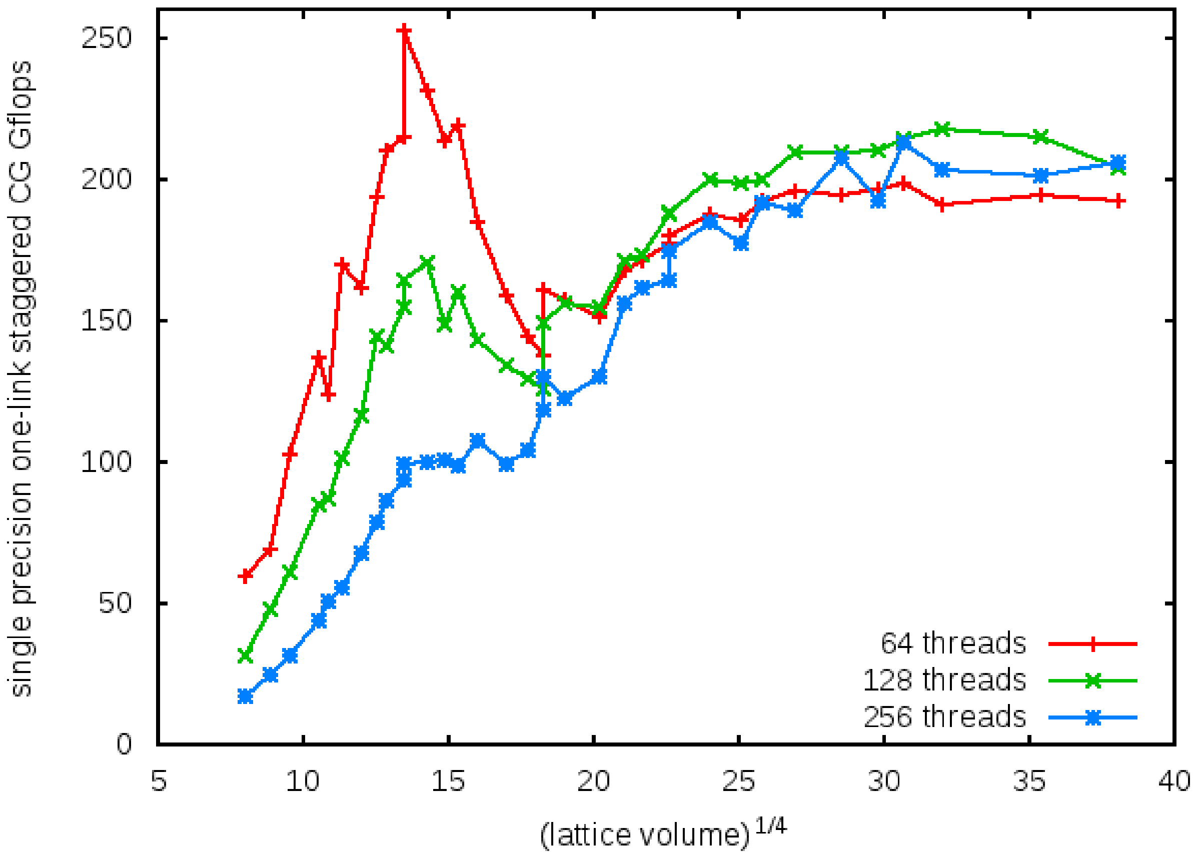
QEX/Nim scripting

- Having scripting interface to application provides:
  - Flexible, procedural, interface to set up parameters
  - Avoids recompiling for simple changes in workflow or need to maintain Makefiles for new codes
  - Enables rapid testing and development by providing high level interface to routines
- Nim provides most of this, except for the actual compiling (so far compile times are a few seconds)
- Could plug in Lua
- Nim provides its own scripting interface (Nimscript)
  - Used in compiler for compile-time evaluation
  - Available to plug in to application and can interface with rest of application

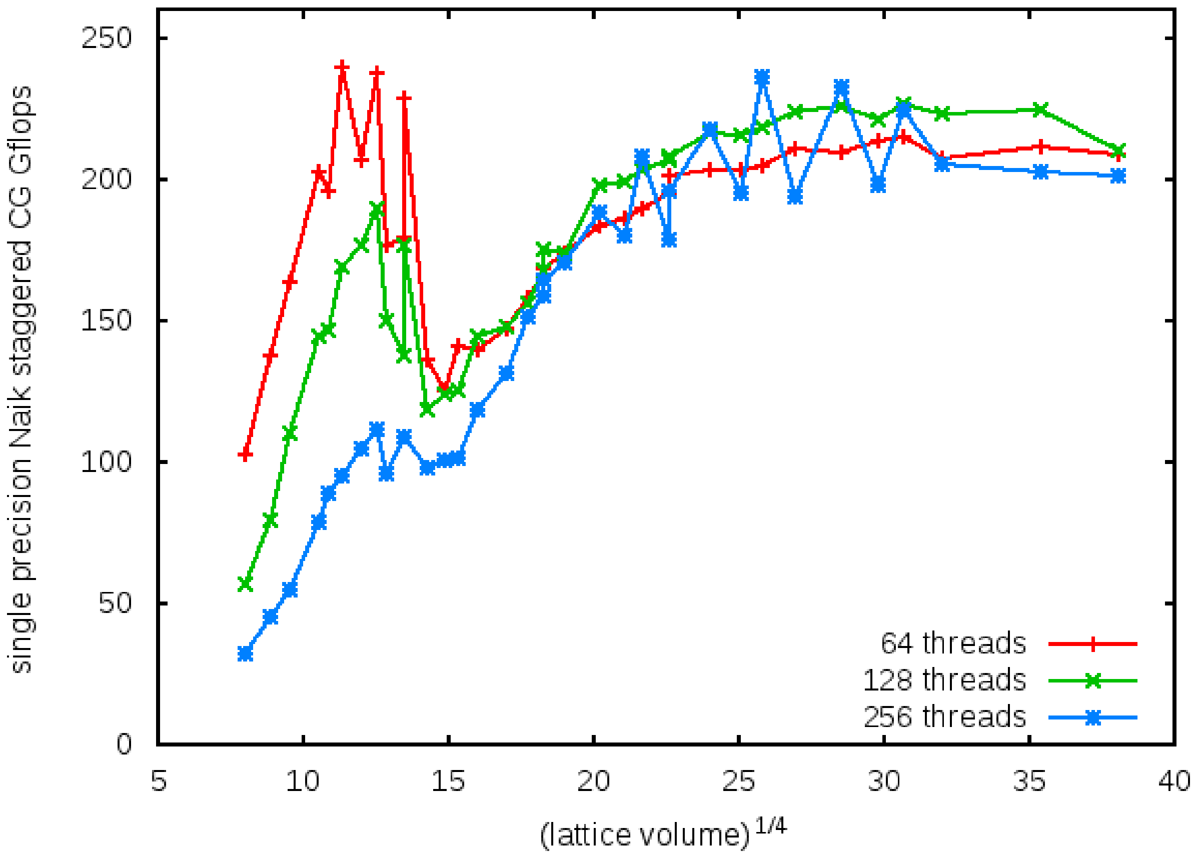
Benchmarks

- Single node KNL Developer Platform
- Intel Xeon Phi CPU 7210
  - 64 cores, 4 hardware threads/core
  - 16 GB high bandwidth memory
- Benchmark staggered CG (with and without Naik term)
- Volumes L^3 x T
  L in {8, 12, 16, 24, 32}
  T in {8, 12, 16, 24, 32, 48, 64}
  with 64, 128 and 256 threads
- Compiled with gcc 6.1
- Plot solver Gflops versus (volume)^(1/4)

Plain (one-link) staggered CG, single precision

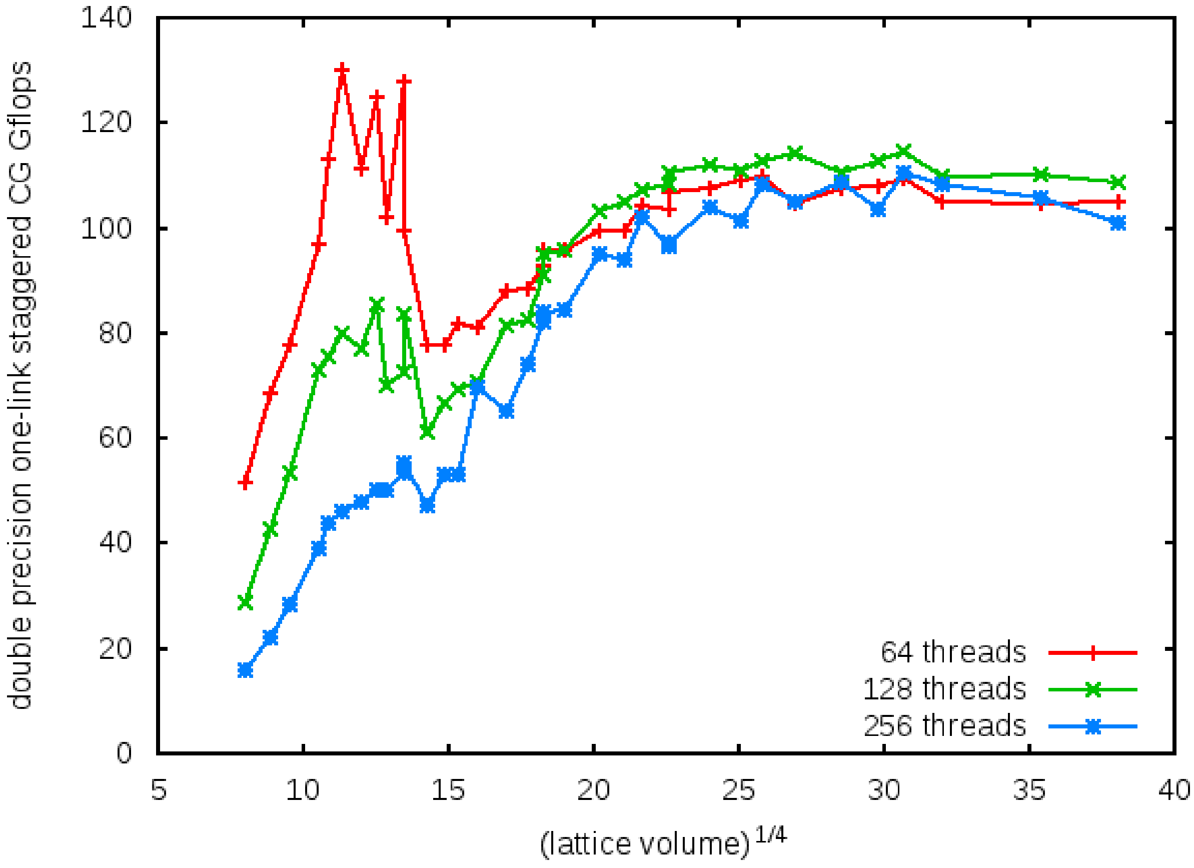


Naik (one-link + three-link) staggered CG, single precision

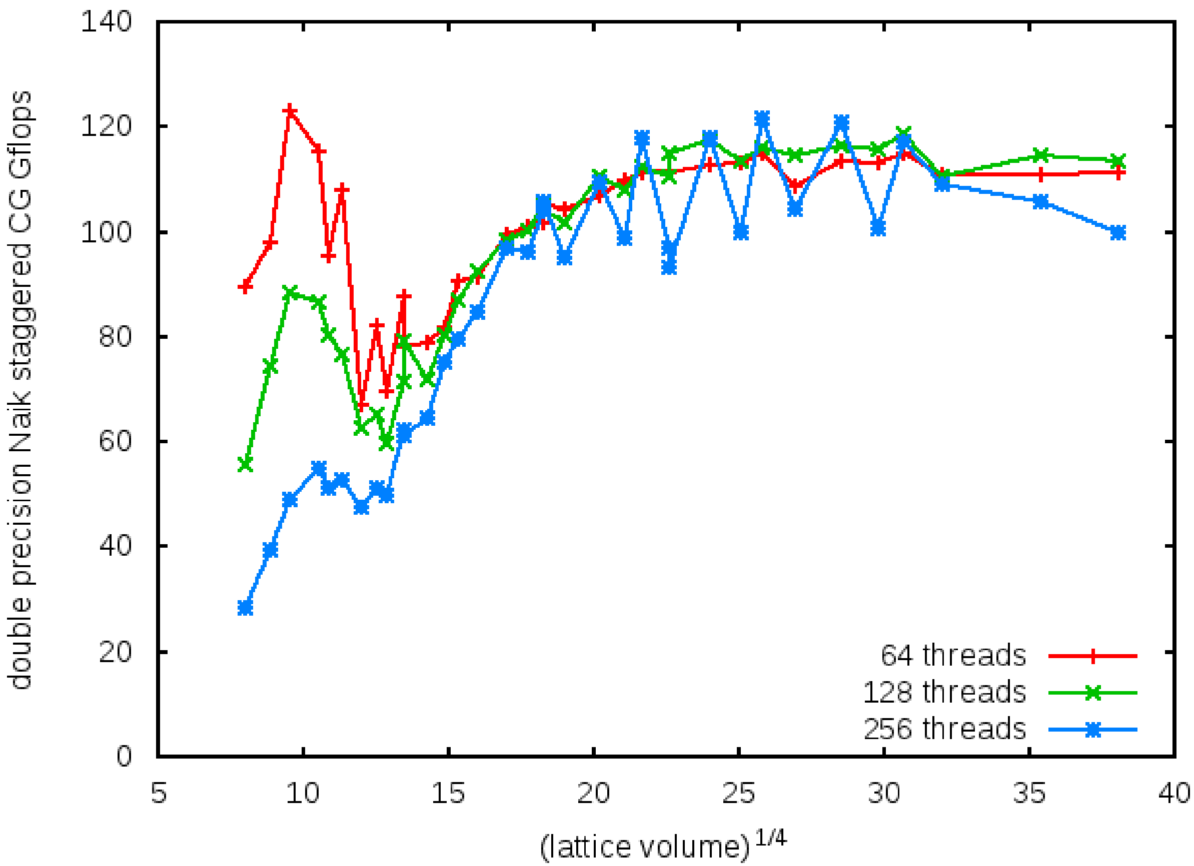


Argonne NATIONAL LABORATORY

Plain (one-link) staggered CG, double precision



Naik (one-link + three-link) staggered CG, double precision



Argonne NATIONAL LABORATORY

Summary

- Nim offers extremely useful set of features
  - Extensive metaprogramming
  - Integrated build system (modules)
  - Simple, high-level “script-like” syntax
  - Seamless integration with C/C++ code, intrinsics, pragmas, etc.
- New QEX framework written in Nim
  - Staggered CG running with good performance on x86 (BG/Q in progress)
  - Working on general optimization framework
    goal: performance portability across compilers & architectures
  - Find more ways to exploit metaprogramming to create easy to use input “languages” for specific operations: smearing, operator contraction, ...

Tuning force-gradient integrators for 8 flavor nHyp HMC

HMC in lattice QCD

- 2 flavor partition function

$$\int dU \mathrm{e}^{-S_g(U)} |D(U) + m|^2$$

- Introduce pseudofermions

$$\int dU d\phi \ \mathrm{e}^{-S_g(U) - \phi^\dagger [(D^\dagger(U) + m)(D(U) + m)]^{-1} \phi}$$

Hybrid Monte Carlo (HMC)
[S. Duane, A.D. Kennedy, B. Pendleton, D. Roweth 1987]

$$\int dp\, dU\, d\phi\ \mathrm{e}^{-p^2/2 - S_g(U) - \phi^\dagger[(D^\dagger(U)+m)(D(U)+m)]^{-1}\phi}$$

- Starting configuration U
- Choose random phi, p
- Evolve U,p according to H from exp(-H(p,U))
- Accept with probability exp(-H(p',U')+H(p,U))

Mass preconditioning [Hasenbusch 2001]

- Introduce extra ratios of determinants

$$|A+\sigma| = \left|\frac{A+\sigma}{A+\sigma_1}\right| \left|\frac{A+\sigma_1}{A+\sigma_2}\right| \dots \left|\frac{A+\sigma_{n-1}}{A+\sigma_n}\right| |A+\sigma_n|$$

$$\phi_1^\dagger \frac{A+\sigma_1}{A+\sigma}\phi_1 + \phi_2^\dagger \frac{A+\sigma_2}{A+\sigma_1}\phi_2 + \dots$$

Choice of integrators

- Perform 'n' steps of a symmetric symplectic integrator
- Leapfrog (XPX) step sizes: (½, 1, ½)ε
- 2force (XPXPX) step sizes: (λ, ½, 1-2λ, ½, λ)ε
  - minimal RMS error coefficients: λ ~ 0.1932
    [I.P. Omelyan, I.M. Mryglod and R. Folk 2003]
    [LQCD: T. Takaishi, P. de Forcrand 2006]
- Force-gradient:
  - Include error term in step

$$\left[\frac{d}{dx_j} S(x)\right] \left[\frac{d}{dx_j}\frac{d}{dx_i} S(x)\right]$$

  - 2 force, 1 gradient (PXGXP) step sizes: (1/6, ½, 2/3, ½, 1/6)
    [M. Suzuki 1995; S.A. Chin 1997]
    [LQCD: A.D. Kennedy, M.A. Clark 2007]
  - 3 force, 1 gradient (XPXGXPX) step sizes: (a0, …, a0)
    a0 = 1/6 [S.A. Chin 1997]
    a0 ~ 0.089 [I.P. Omelyan, I.M. Mryglod and R. Folk 2002]

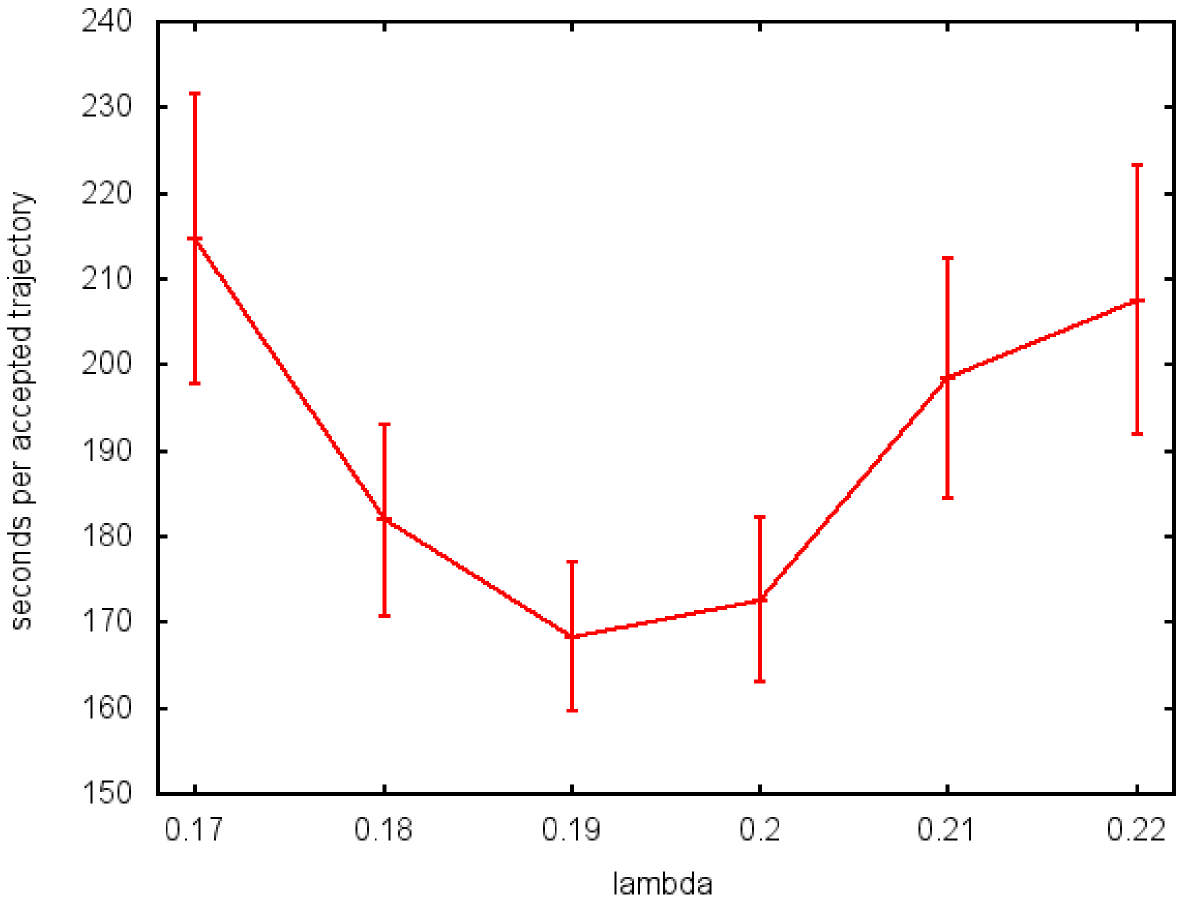
8 flavor nHyp simulation setup

- Using LSD collaboration 8 flavor nHyp ensembles as test of integrators
- 24^3 x 48
- m = 0.00889
- Using 9 Hasenbusch mass preconditioners
- Fermion integrator using 24 force steps per trajectory
- Using FUEL (very flexible integrator framework)

- Test integrators using "seconds / accepted trajectory" as metric

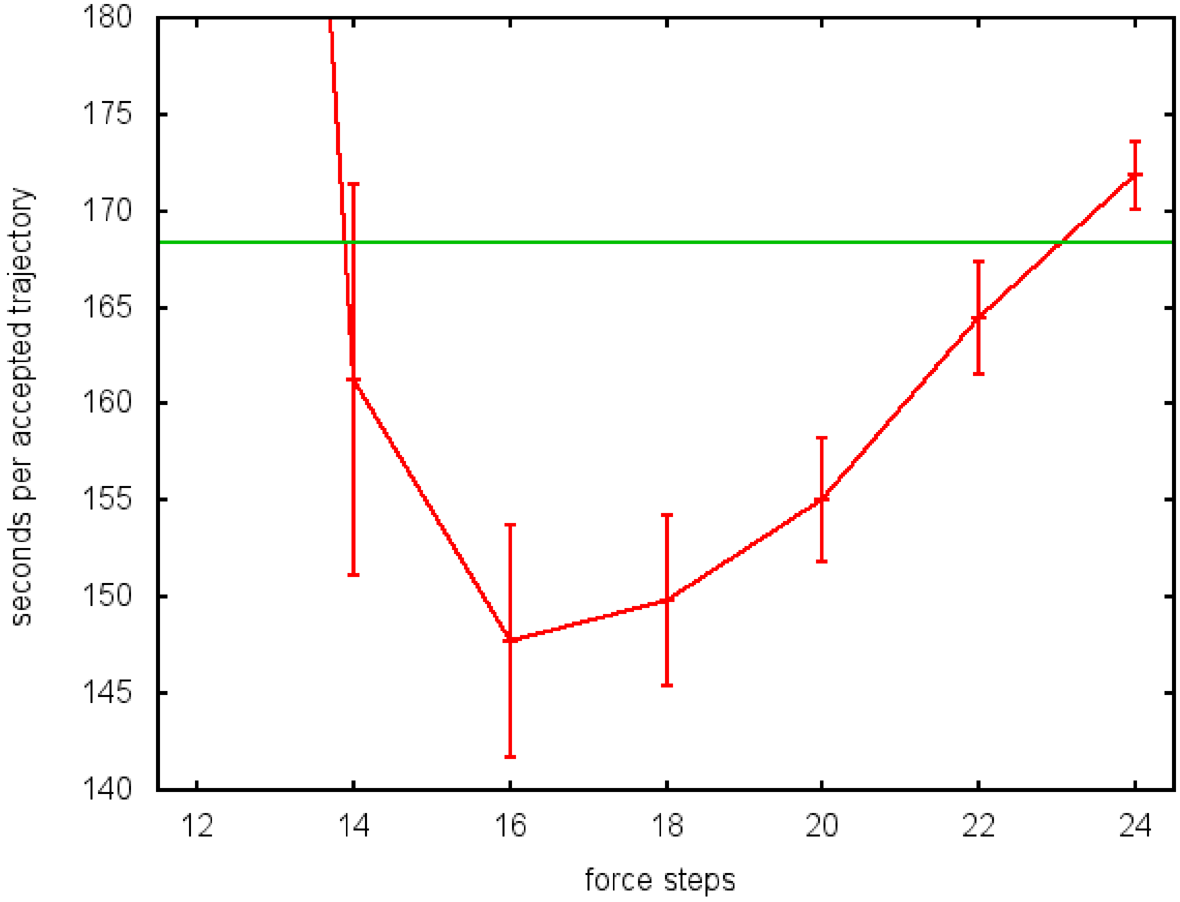
$$\frac{\text{trajectory seconds}}{\text{accpetance rate}}$$

2 force (XPXPX) integrator



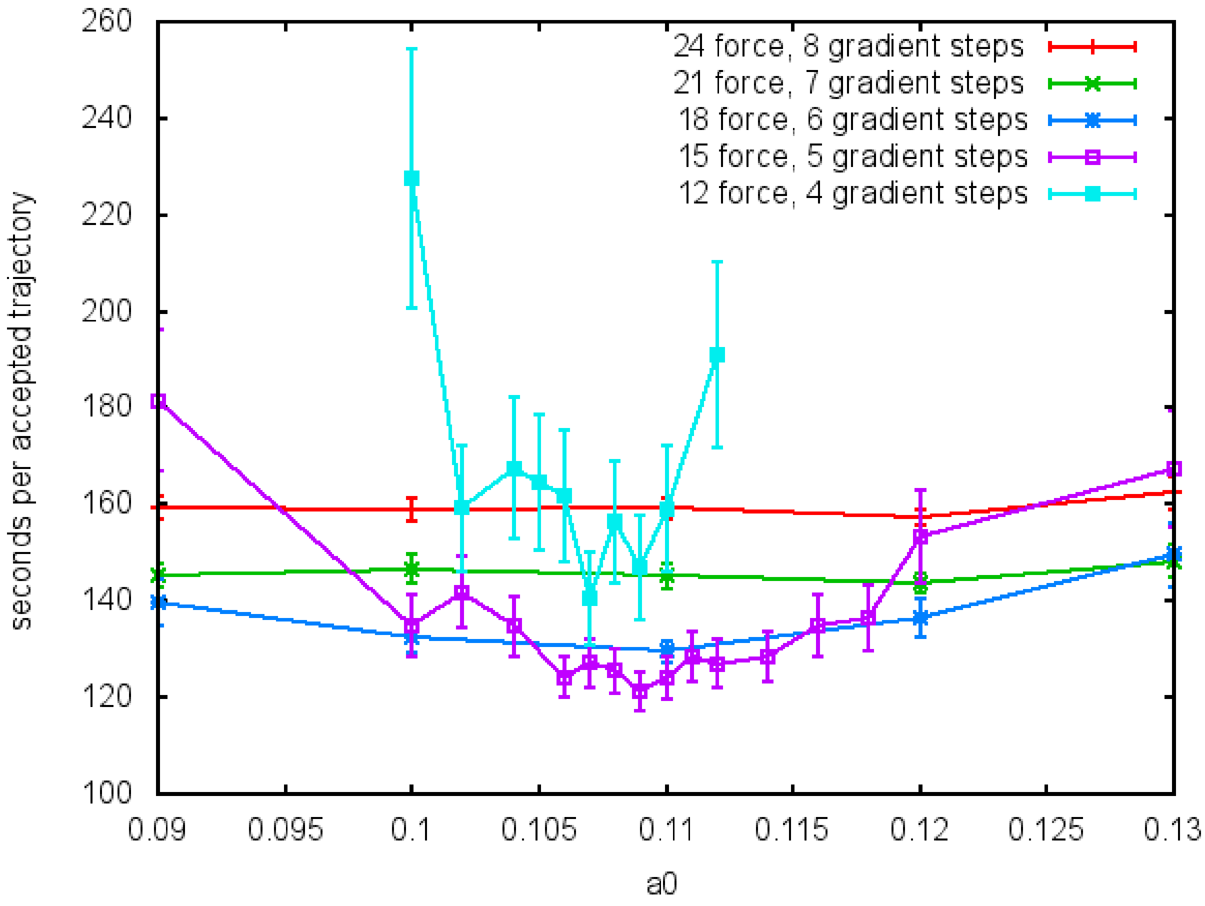
Argonne NATIONAL LABORATORY

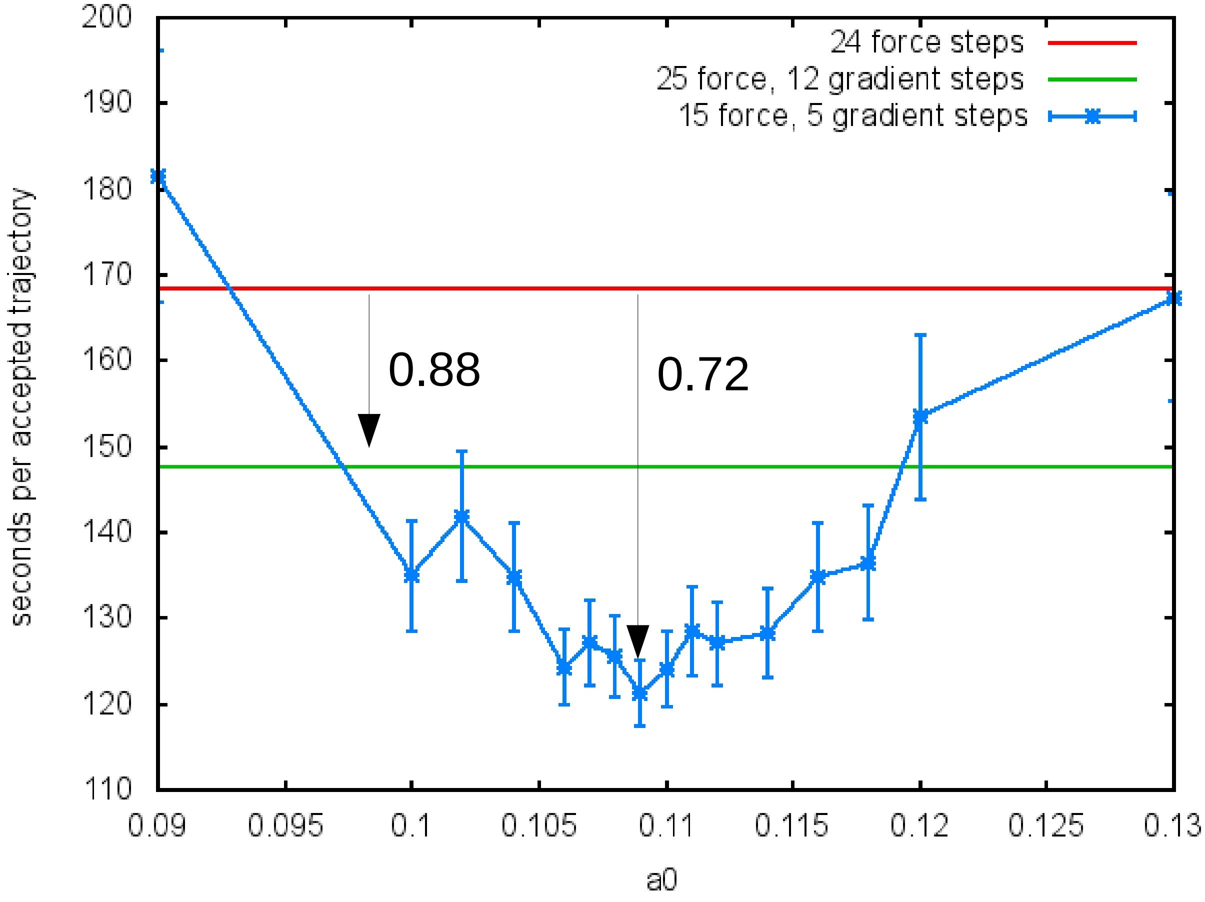
2 force, 1 gradient (PXGXP)



Argonne NATIONAL LABORATORY

3 force, 1 gradient (XPXGXPX)





24 force steps
25 force, 12 gradient steps
15 force, 5 gradient steps
0.88
0.72
seconds per accepted trajectory
a0
200
190
180
170
160
150
140
130
120
110
0.09
0.095
0.1
0.105
0.11
0.115
0.12
0.125
0.13

Summary

- Lots of possibilities for integrator patterns
- Longer patterns can give improvement when tuned well