Simulating Ultralight Dark Matter with Chapel

An Experience Report

Nikhil Padmanabhan [1] Elliot Ronaghan [2]
J. Luna Zagorac [1] Richard Easther [3]

2019/11/17

[1]Yale Univ.

[2]Cray

[3]Univ. of Auckland

Table of contents

Introduction

Motivating Ultralight Dark Matter

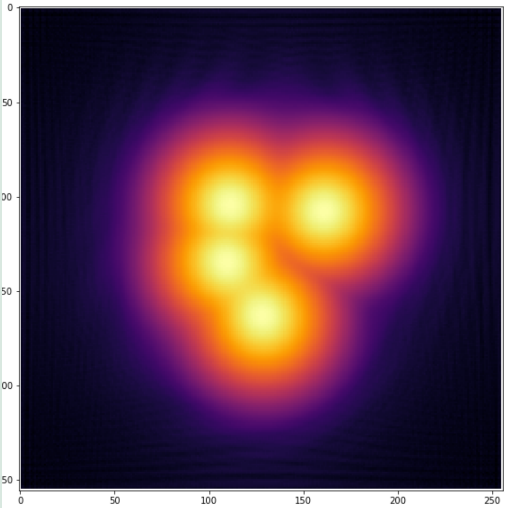
- In the standard cosmological model, 80% of the matter in the Universe is "dark" (i.e. non-baryonic).
- Explains a large scale of observations, from the rotation of galaxies, to "Bullet" clusters, to the distribution of galaxies, to the cosmic microwave background.
- Form gravitationally bound structures : dark matter halos.
- The traditional model is a heavy particle ($\sim 100\times$ proton), with weak interactions.
- Possible puzzles remain on small scales from the structure of dark matter halos, to the observed abundance of dark matter halos. Note that these might well be solved by astrophysics.
- We have not detected these in the lab, or at accelerators.

Motivating Ultralight Dark Matter

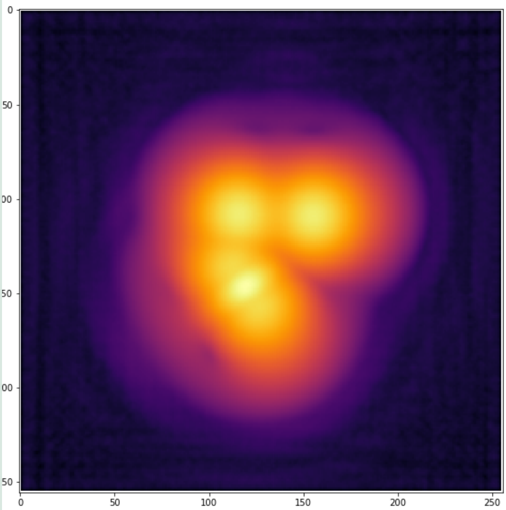
- A different paradigm is a very light particle ($10^{-31}\times$ proton).
- Many names : fuzzy dark matter, Bose-Einstein dark matter, ...
- Small mass means that quantum-mechanics can smear it out over astrophysically interesting scales.
- High enough density that it forms a Bose-Einstein condensate.
- Different phenomenology : eg. interference patterns.
- Anything by the most idealized situations requires simulations.

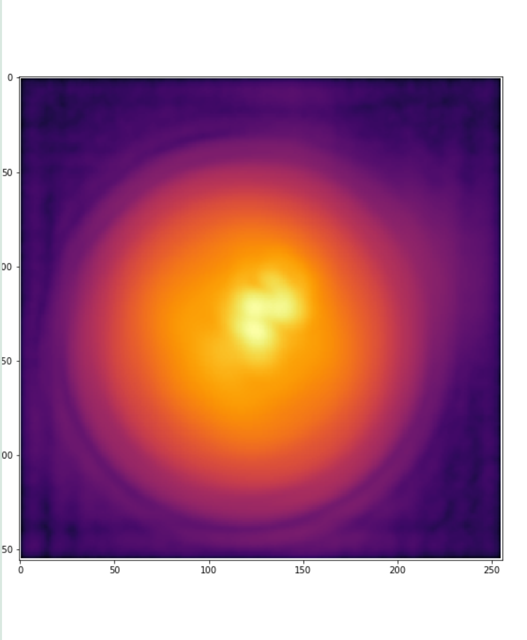
Our Motivation

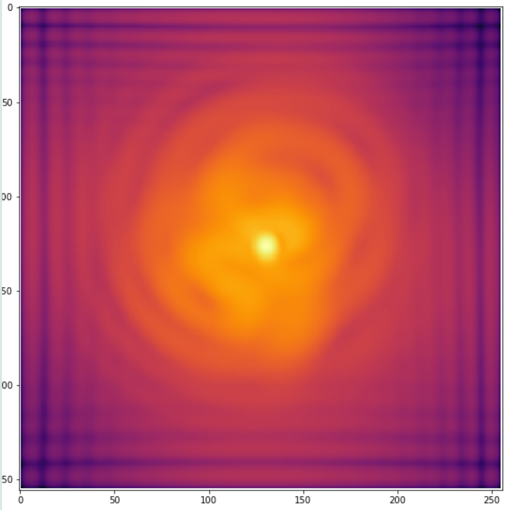
- Want a code to do numerical experiments with.
- Need scalability
  - Must resolve soliton cores : large dynamic range
  - Simulation time scales as N^5; need to scale to large numbers of nodes.
- Initial problem : revisit aspects of the formation of ultra-light dark matter halos from collisions of soliton cores.
- This is an area of very active research (we are newcomers).
- Several codes exist - including adaptive codes, codes built on existing large astrophysical simulations. Challenges to large boxes still exist.
- An incomplete list : Schive et al, 2014, Mocz et al, 2017, Edwards et al 2017, Veltmaat et al, 2018



0
50
00
50
00
50
0
50
100
150
200
250


0
50
00
50
00
50
0
50
100
150
200
250


0
50
00
50
00
50
0
50
100
150
200
250


0
50
00
50
00
50
0
50
100
150
200
250

History of Project

- `PyUltraLight`[a]: An initial code in Python, driven by Jupyter notebook
  - Easy to use and modify, allowing numerical experiments
  - Performant and multithreaded (made significant use of eg. `numexpr`, `FFTW`)
- Extending to isolated potentials hit Python bottlenecks
- Attempted a skunkworks (2019/6/22) port to Chapel for a single node. Resulting code not much longer than Python, could implement isolated potentials, better multithreaded performance.
- Distributed Code
  - Want to run larger N_{grid}, can we extend the code?
  - Isolated potential calculation led to wanting a native Chapel distributed FFT (useful for many other tasks).[b]
  - Validating the FFT led to the NAS NPB benchmark.

[a] Edwards et al, arXiv:1807.04037

[b] Note that Chapel can also interoperate with MPI.

The Schrodinger-Poisson Equations

The Model

$$i\hbar\frac{\partial\psi}{\partial t} = -\frac{\hbar^2}{2m}\nabla^2\psi + m\Phi\psi$$
$$\nabla^2\Phi = 4\pi G m|\psi|^2$$

Timestepping

$$\psi(\mathbf{x}, t+h) \approx \exp\left[\frac{-ih}{2}\Phi(\mathbf{x}, t+h)\right]\exp\left[\frac{ih}{2}\nabla^2\right]\exp\left[\frac{-ih}{2}\Phi(\mathbf{x}, t)\right]\psi(\mathbf{x}, t)$$

Taking Fourier transforms simplifies the kinetic term

$$\psi(\mathbf{x}, t+h) \approx \exp\left[\frac{-ih}{2}\Phi(\mathbf{x}, t+h)\right]\mathcal{FT}^{-1}\exp\left[\frac{\mathrm{ih}}{2}\mathrm{k}^2\right]\mathcal{FT}\exp\left[\frac{-ih}{2}\Phi(\mathbf{x}, t)\right]\psi(\mathbf{x}, t)$$

See eg. Edwards et al, arXiv:1807.04037

FFTs in Chapel

Slab Decompositions Are Simple

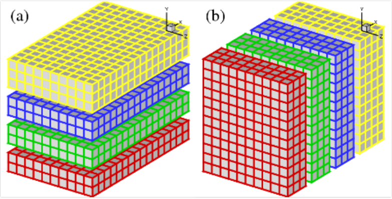


Figure 1: Slab Decomposition

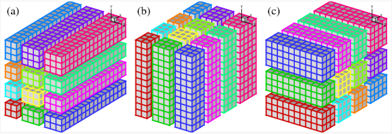


Figure 2: Pencil Decomposition

`http://www.2decomp.org/decomp.html`

- Slab decompositions are simpler (especially for the end user)
- Slab limits the amount of parallelism expressed (especially with pure MPI)
- Use 1 slab per locale/node.
- Limits $N_{\mathrm{grid}} \geq N_{\mathrm{nodes}}$, but in practice, not limiting.
- Reduce communication complexity

Chapel Code is Expressive : Pencil and Paper

The Algorithm

1. Decompose array into slabs in the *x* direction
2. Fourier transform in the *y* direction[1]
3. Fourier transform in the *z* direction
4. Transpose *x* and *y* (all to all)
5. Fourier transform in the *x* direction

[1] We use FFTW (`www.fftw.org`) for 1D serial transforms.

Chapel Code is Expressive : A Naive Implementation

```
coforall loc in Locales do on loc {
  ...
  for ix in xSrc {
    myplane = Src[{ix..ix, ySrc, zSrc}];
    // Y-transform
    forall iz in zSrc {
      yPlan.execute(myplane[0, ySrc.first, iz]);
    }
    // Z-transform, offset to reduce comm congestion/collision
    forall iy in offset(ySrc) {
      zPlan.execute(myplane[0, iy, zSrc.first]);
      // Transpose data into Dst
      Dst[{iy..iy, ix..ix, zSrc}] = myplane[{0..0, iy..iy, zSrc}];
    }
  }
  // Wait until all communication is complete
  allLocalesBarrier.barrier();
  // X-transform, similar to Y-transform
  ...
}
```

Chapel Code is Expressive : A Naive Implementation

```
coforall loc in Locales do on loc {
  ...
  for ix in xSrc {
    myplane = Src[{ix..ix, ySrc, zSrc}];
    // Y-transform
    forall iz in zSrc {
      yPlan.execute(myplane[0, ySrc.first, iz]);
    }
    // Z-transform, offset to reduce comm congestion/collision
    forall iy in offset(ySrc) {
      zPlan.execute(myplane[0, iy, zSrc.first]);
      // Transpose data into Dst
      Dst[{iy..iy, ix..ix, zSrc}] = myplane[{0..0, iy..iy, zSrc}];
    }
  }
  // Wait until all communication is complete
  allLocalesBarrier.barrier();
  // X-transform, similar to Y-transform
  ...
}
```

- Can use SPMD programming when needed
- Data parallel features
- PGAS - simple communication
- User-defined iterators (eg. offset)

Chapel Code is Expressive : A Performant Implementation

```
...
forall iy in offset(ySrc) {
  zPlan.execute(myplane[0, iy, zSrc.first]);
  // Transpose data into Dst, and copy the next Src slice into myplane
  copy(Dst[...], myplane[...], myLineSize);
  if (ix != xSrc.last) {
    copy(myplane[...], Src[...], myLineSize);
  }
}
...
```

- Overlap computation and communication, even with blocking comm
- Use low-level communication primitives

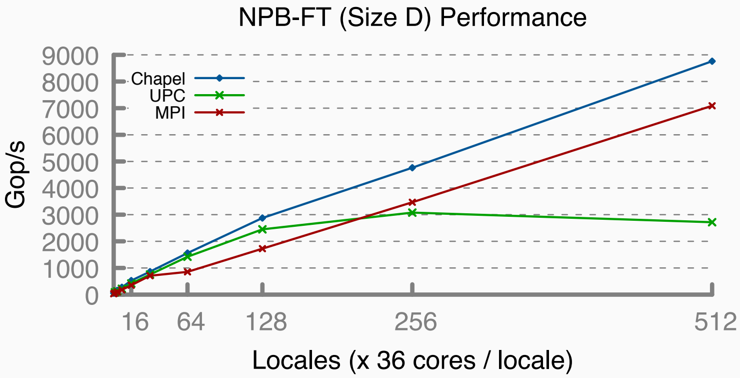
Machine/Compiler Specifications

- Scalability Hardware (Cray-XC):
  - 36-core (72 HT), 128 GB RAM
  - dual 18-core (36 HT) ”Broadwell” 2.1 GHz processors
- Software
  - CLE 7.0.UP01
  - Intel Compilers 19.0.5.281
  - FFTW 3.3.8.4
  - Chapel 1.20.0
  - Cray 9.0.2 (classic)

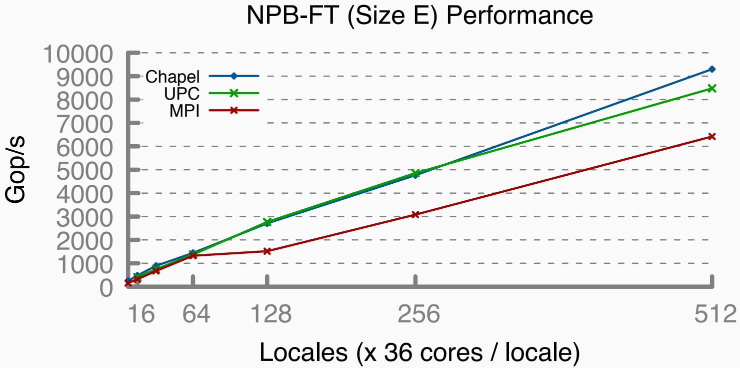
Benchmarks

- Use the NAS NPB-FT benchmark
  - NPB v3.4
  - Class D ($2048 \times 1024 \times 1024$), E ($8\times$), F($8\times$)
- Compare Chapel, MPI reference and UPC (with non-blocking overlapped comm)
- MPI and UPC use pencil decompositions for large problems/node counts.
- MPI and UPC use 32 cores/node (require a power of 2)
  - Restricting Chapel to 32 cores does not significantly change timings, indicating memory/communication bound.

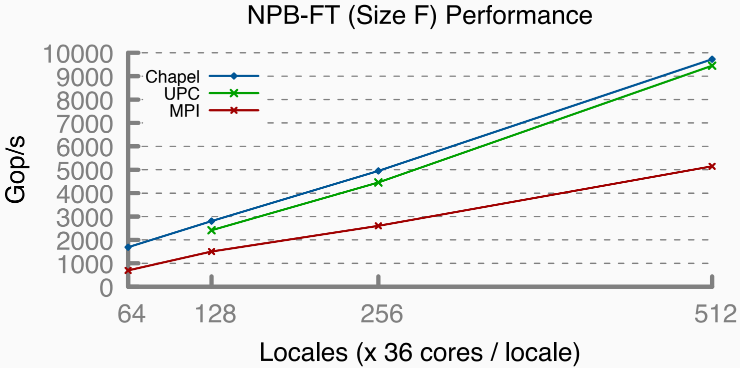
Chapel FFTs Scale Well Across Nodes : Class D



Chapel FFTs Scale Well Across Nodes : Class E = 8×D

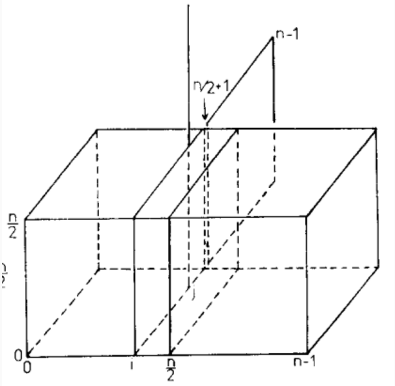


Chapel FFTs Scale Well Across Nodes : Class F = 8$\times$E



Isolated Gravitational Potentials

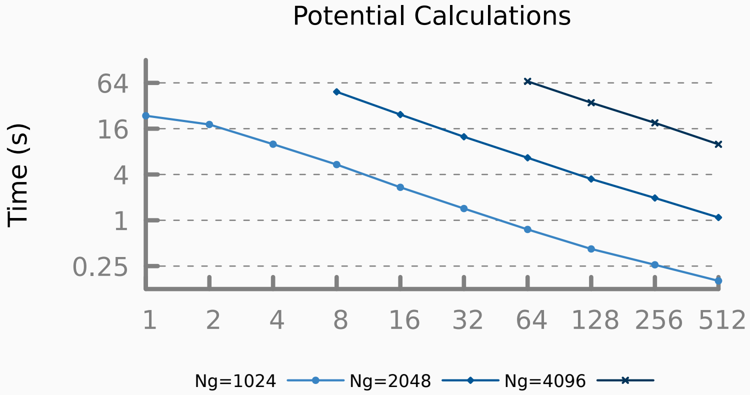
Calculating Potentials with Isolated Boundary Conditions



- Standard solution uses Fourier transforms, but assumes periodic boundary conditions.
- Need isolated boundary conditions $\Phi(\infty) = 0$.
- Convolve with $1/r$ Green's function, pad domain with zeros.
- Use separability of Fourier transforms to do each dimension separately.
- Only doubles the computational domain, not a factor of 8.

Hockney, 1970 Eastwood & Brownrigg, 1979

Potential Calculations Scale Well



The Full Simulation

Going to Distributed Code

Setting the boundary of the domain

```
iter boundary(d : domain, thick : int=1) {
  param rank = d.rank;
  for param idim in 1..rank {
    var off : rank*int;
    off(idim)=thick;
    yield d.interior(off);

    off(idim)=-thick;
    yield d.interior(off);
  }
}
```

- Same code as single locale version
- Sometimes, things work seamlessly!
- Highlights strength of Chapel domains

Going to Distributed Code

Initialization Code : Single-Node Version

```
proc addSoliton(..., psi : [?Dom] complex) {
  ...
  forall ijk in Dom {
    ...
    psi[ijk] += val*phase1*phase2;
  }
}
```

Initialization Code : Distributed Version

```
proc addSoliton(..., psi : [?Dom] complex) {
  ...
  forall ijk in Dom with (var myprofile = this.profile,
                          ...)
  {
    ...
    psi.localAccess[ijk] += val*phase1*phase2;
  }
```

Going to Distributed Code

Initialization Code : Single-Node Version

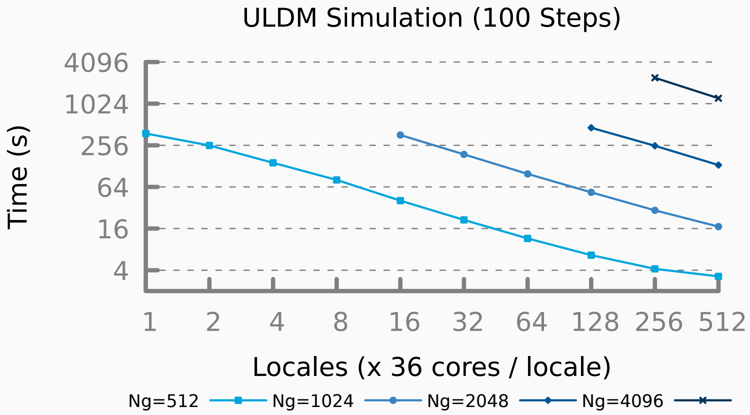
```
proc addSoliton(..., psi : [?Dom] complex) {
  ...
  forall ijk in Dom {
    ...
    psi[ijk] += val*phase1*phase2;
  }
}
```

Initialization Code : Distributed Version

```
proc addSoliton(..., psi : [?Dom] complex) {
  ...
  forall ijk in Dom with (var myprofile = this.profile,
  ...)
  {
    ...
    psi.localAccess[ijk] += val*phase1*phase2;
  }
```

- Need to localize some variables (sometime will be surprised by comm)
- The compiler currently does not fully optimize local array accesses.
- Note that `psi` changes from a local to a distributed array, but this does not affect eg. the `forall` loop.

Simulations Scale Well



Lessons Learned

Where Chapel could do better

1. Tooling
   - Identifying communication - how much and from where? How to recognize a sub-optimal pattern.
   - Easier profiling
   - Compiler improvements, including speed.
2. Easier to express low-level communication/locality
   - Low level communication primitives are not exposed to user (useful when the user can reason better about the communication patterns).
   - Verbose to express locality of computation and have the compiler optimize appropriately.
3. Fewer hidden performance traps
   - Unexpected communication
   - Promotion of operations over N-d arrays can be slow.

None of these are new issues to the Chapel team (and many have Github issues).

My Thoughts

- HPC:
    - *Productivity*: Chapel design has scientific codes in mind.
        - Domains/Arrays
        - Expressive Parallelism - where/when you need it.
        - Interoperability - C (and now Python!)
    - *Performance*: Chapel code can perform/scale very well without heroic efforts.
- It's a fun language to write. Easy to throw together prototype code in. And it largely does the right thing!
- I'm getting to the point where I could imagine just working in Chapel. And I'm more open to getting students to work in it.
- This is the first code that I've had run on ∼18K cores. Significant credit for that goes to Elliot (and the Chapel team), but the language played a non-trivial role here.

Backup Slides

Chapel FFTs Scale Well Across Nodes : Class D