Simulating Ultralight Dark Matter with Chapel : An Experience Report

Nikhil Padmanabhan
Dept. of Physics, Yale Univ.
New Haven, CT, USA
nikhil.padmanabhan@yale.edu

Elliot Ronaghan
Cray Inc.
Seattle, WA, USA
eronagha@cray.com

J. Luna Zagorac
Dept. of Physics, Yale Univ.
New Haven, CT, USA
jovana.zagorac@yale.edu

Richard Easther
Dept. of Physics, Univ. of Auckland
Auckland, New Zealand
r.easther@auckland.ac.nz

***Abstract*—We describe our implementation of an astrophysical code to simulate the dynamics of ultralight dark matter in Chapel. We focus on the programmability of Chapel, highlighting the relative ease of translating the physics of this system into a code that can run efficiently on distributed systems. We also demonstrate that this code can scale well from small problem sizes that can be run on laptops to large problem sizes run across hundreds of processors. We finally present the results from a few simulations of astrophysical interest. An interesting by-product of broader interest is a distributed FFT routine written in Chapel; we summarize its implementation and performance.**

I. Extended Abstract

Modern cosmology gives a coherent description of the evolving universe from the first instants after the Big Bang through to the present day. In this framework, dark matter drives the formation of galaxies, controlling their overall evolution and internal dynamics, and accounts for 25% of the current universe by mass. Dark matter is thus a dramatic success for cosmology and a profound challenge for particle physics: it explains key properties of the universe and makes well-verified predictions but its fundamental composition is mysterious and entirely unknown. Many dark matter models exist and WIMPs (Weakly Interacting Massive Particles) are the most widely studied. WIMP-like particles are found in plausible particle physics theories but it is crucial to consider a range of options, especially as WIMPs face increasingly tight constraints from Large Hadron Collider data and direct detection experiments, along with nagging discrepancies with the small-scale (astronomically speaking) properties of galaxies. Ultralight (also fuzzy- or axion-) dark matter (ULDM) is a fascinating possibility which is receiving intense attention from the worldwide cosmology community (see [1] for a recent review). Ultralight dark matter naturally smooths small-scale astrophysical structure, ameliorating a key shortcoming of WIMP models, while being compatible with many candidate "theories of everything" including string theory.

In order to make predictions for the phenomenology of ULDM requires numerical simulations, especially in order to capture the unique interference effects that can arise from its wave-like nature. These simulations can be quite challenging, since they must resolve a large dynamical range of scales. Some examples of recent codes are [2]–[4]; [5] provides a recent review of the numerical approaches used.

Our starting point is [4], which is a pseudo-spectral code written in Python. One of the advantages of being in Python is its flexibility, making it easy for researchers to experiment with various extensions. However, this Python code has a few disadvantages - (1) it is hard to extend to a distributed computing environment and (2) while it calls into highly tuned numerical libraries [6] and makes use of JIT compilation [7], it is still easy to be limited by Python's speed while extending the code. Avoiding these shortcomings without losing the expressiveness of Python is what drew us to rewriting this code in Chapel. This talk will discuss our experiences from this process, highlighting both successes and challenges.

Outline: An outline of our proposed talk is as follows : we start with a brief discussion of the underlying physics of ultralight dark matter and our approach to simulating it. We then turn to an implementation of a distributed fast Fourier transform in Chapel, highlighting the advantages of the "multi-resolution" and PGAS nature of Chapel. We will present the results from some simulations of astrophysical interest as well as performance studies of the code. Finally, we will conclude with a discussion of the lessons learned as well as future directions. The remainder of this abstract expands on this work in progress in more detail, presenting preliminary results.

The Physics: Since our implementation derives directly from [4], we refer to that work for a detailed discussion of the exact algorithms used and just provide a brief summary here. The starting point for simulating ULDM is the Schrodinger-Poisson equations (written below in dimensionless code units):

$$i\dot{\psi} = \frac{-1}{2}\nabla^2\psi + \Phi\psi \tag{1}$$

$$\nabla^2\Phi = 4\pi|\psi|^2 \,. \tag{2}$$

The first of these is the Schrodinger equation which describes the time evolution of the complex wavefunction ψ, while the second is the Poisson equation that determines the gravitational potential Φ sourced by the density given by $|\psi|^2$. Following [4], the time evolution can be written as a second-order symmetrized split-step (leap-frog) operator

$$\psi(t+h) \approx \exp\left[\frac{ih}{2}\Phi(t+h)\right]\exp\left[\frac{ih}{2}\nabla^2\right]\exp\left[\frac{-ih}{2}\Phi(t)\right]\psi(t) \tag{3}$$

where we have suppressed the position dependence for simplicity. The two outermost operations (involving Φ) are most directly written in configuration space, while the middle equation is directly expressed in Fourier space. This makes a pseudo-spectral approach natural for this problem.

We diverge from [4] in our calculation of the gravitational potential. [4] chooses periodic boundary conditions which allows the Poisson equation to be directly solved using Fourier methods. However, these periodic boundaries can introduce numerical artifacts, and so we choose to determine the potential by treating our simulation box as being isolated. This is equivalent to convolving the source function by the $1/r$ Green's function of gravity; these convolutions may be efficiently done using Fourier transforms. However, to avoid the effect of periodicity, we must pad the box, effectively working on a $(2N)^3$ grid. We do so using the method of [8], which uses the separability of the Fourier transform to intersperse the convolution steps within the Fourier transform and only requires a $2N^3$ grid (a considerable savings in memory). [1]

Distributed FFTs: The principal element behind pseudo-spectral codes is the Fast Fourier Transform (FFT). When considering distributed Fourier transforms in Chapel, we had two separate routes available to us. The first was to use Chapel's interoperability with MPI and call into a standard MPI based multidimensional FFT routine like FFTW [9] or PFFT [10]. However, as discussed previously, steps such as the gravitational potential calculation require interspersing additional computations in between the one-dimensional Fourier transforms that make up the full three-dimensional transform. This meant that we could no longer simply call into a single MPI routine, removing the convenience offered by this approach. We therefore opted for the second alternative of implementing the three-dimensional FFT directly in Chapel.

Our three-dimensional relies on the optimized FFTW library [9] for the local transforms (which incidentally highlights Chapel's interoperability with existing libraries); we therefore only use Chapel to reorganize the distributed data to present a local view to the FFTW routines. For simplicity, we adopt a one-dimensional slab decomposition, where the x dimension is distributed across multiple nodes (referred to as *locales* in Chapel, while the y and z dimensions of the grid are local to each node. As is possible with many FFT algorithms, we leave the array in a transposed state after the transform. Therefore, the destination array is stored as $y{-}x{-}z$, with the y dimension first and distributed across multiple locales, while x and z now locally stored across each grid. The actual 1D FFTs along each of the axes are easily performed by direct calls to FFTW. We use the single threaded FFTW routines, and use Chapel's task parallelism to run on multiple cores. This avoids mixing different threading runtimes, and can achieve similar performance with an appropriate batching of the FFTW calls. Motivated by prior studies of FFT performance in UPC [11], we perform $x \leftrightarrow y$ transpose row by row[2], transmitting each row as soon as the Fourier transform in that direction is complete. While this allows for good performance (see below) this code does not do the FFT in-place (unlike both [9] and [10]), but uses a second work/destination array. Although generally a disadvantage, this does not significantly impact our simulation code, since it requires multiple arrays, one of which can be used as the work array.

A part of the actual Chapel code, highlighting the yz transform and transpose is in Listing 1. It is worth highlighting a few aspects of this code. The first is the expressiveness of the PGAS approach and one-sided communication. The array `arr` is distributed, and yet the Chapel code closely follows the pen-and-paper described above and is not cluttered with explicit communication/parallelism calls.

Both the data-parallel (the `forall` loop) and the task-parallel aspects (the `coforall` loop) of Chapel are also seen in the code above. We use the data-parallel constructs to express the on-node parallelism implicitly, while we use the explicit task-parallel constructs to write a more traditional SPMD code. It's worth noting that the code above could be written in a purely implicit data-parallel approach (and we do so in a number of places in our code), or in a purely explicit task-parallel approach. The fact that Chapel allows the user to mix these (either for performance or expressiveness) is a major strength of the language.

The simplicity of the above code also makes extending it for the gravitational potential calculation straightforward. Indeed, the actual gravitational potential code follows exactly the same structure as the FFT code.

While the above code highlights the expressiveness of Chapel and its productivity for the end-user, it is important to also consider the performance of this code. To this end, we implemented NPB-FT benchmark (version 3.4) [12] in Chapel. Fig. 1 shows the performance for the Class D ($1024\times1024\times2048$ grids), E ($2048\times2048\times4096$ grids) and F ($4096\times4096\times8192$ grids) problem sizes for increasing node count. The figure also compares the Chapel performance with the reference implementation provided with the benchmark. All of these tests were run on a Cray XC system with 2.1 GHz Broadwell nodes with 36 cores and 128 GB of RAM. The Chapel codes were compiled with the 1.20 release version of the compiler, and standard optimization flags were used in both cases.

Unlike the reference implementation, the Chapel implementation uses the same 1D data decomposition for all problem sizes and node counts. This is not possible in a pure MPI code, since the number of MPI ranks must be less than the the size of the dimension one is distributing. In Chapel, this restriction is on the number of nodes, allowing for a much larger number of nodes. Chapel's task-parallel features (both local and distributed) allow the user to express the parallelism in the problem. Finally, the PGAS nature of the language simplifies data movement across the nodes. As Fig. 1 shows, the expressiveness of the Chapel code does not necessarily come with a performance cost, with the Chapel version competitive with the reference version across problem sizes and node counts.

[1]It was the challenge of efficiently implementing this step in Python that originally led us to exploring the use of Chapel.

[2]where "row" here refers a varying z index

```
// Start of SPMD section.
coforall loc in Locales {
  on loc {
    // Get array dimensions, and set up FFTW plans
    // The local part of the array arr has dimensions
    // arr[xRange, yRange, zRange]
    // Declarations and initial set up elided for brevity.

    // Transform the y-dimensions first
    // Do this in parallel
    forall (ix, iz) in {xRange, zRange} {
      // Get reference to start of y dimension
      var elt = c_ptrTo(arr.localAccess[ix, y0, iz]);
      // Execute FFTW plan
      plan_y.execute(elt, elt);
    }

    // Do the z transforms
    forall (ix, iy) in {xRange, yRange} {
      // FFT the z row
      var elt = c_ptrTo(arr.localAccess[ix, iy1, z0]);
      plan_z.execute(elt, elt);

      // Put the data into the destination array,
      // transposing x and y. This could be written more elegantly as
      // dest[iy,ix,..] = arr[ix,iy,..]
      // However, the version below is more performant
      ref srcRef = arr.localAccess[ix, iy, z0];
      ref dstRef = dest[iy,ix,z0]; // Get reference to remote data
      // Chapel doesn't currently expose a user-level put, so we need to
      // call into a primitive.
      __primitive("chpl_comm_put", srcRef, dstRef.locale.id, dstRef, myLineSize);
    }
  }

  // End of SPMD section
  }
}
```

Listing 1: Our implementation of the transform in the yz direction including the transpose. The advantages of PGAS and one-sided communication are clearly evident from the simplicity of the code. Although the code above does perform well, it incurs a few overheads from the repeated FFTW calls. Our final timings avoid these by batching a number of the FFTW calls together. While this loses some of the elegance of this code, the complexity of the code is not significantly affected.

Note that we do not attempt any optimization of the NPB benchmark, nor is this meant as an exhaustive comparison of different distributed FFT codes. Rather, we present our results as a demonstration of performance achievable in Chapel.

Although the timing results shown here are for the FT benchmark, the full workload for the ULDM simulation is very similar and therefore, we anticipate getting similar scaling results for the full code.

Questions: We conclude with a series of questions we aim to answer in the talk:

1) *How expressive/productive is Chapel?* We have already provided a partial answer to this question, but will demonstrate with more examples from the code. We also will compare this to the corresponding Python code from [4].
2) *How performant is Chapel code?* We have used the NPB-FT benchmark as a useful proxy code, but we will show more detailed timing results from the full ULDM code.
3) *What are the challenges to coding in Chapel?*
4) *Further extensions to this code?*

Acknowledgments: We thank the Cray Chapel team for useful discussions and support of this work.

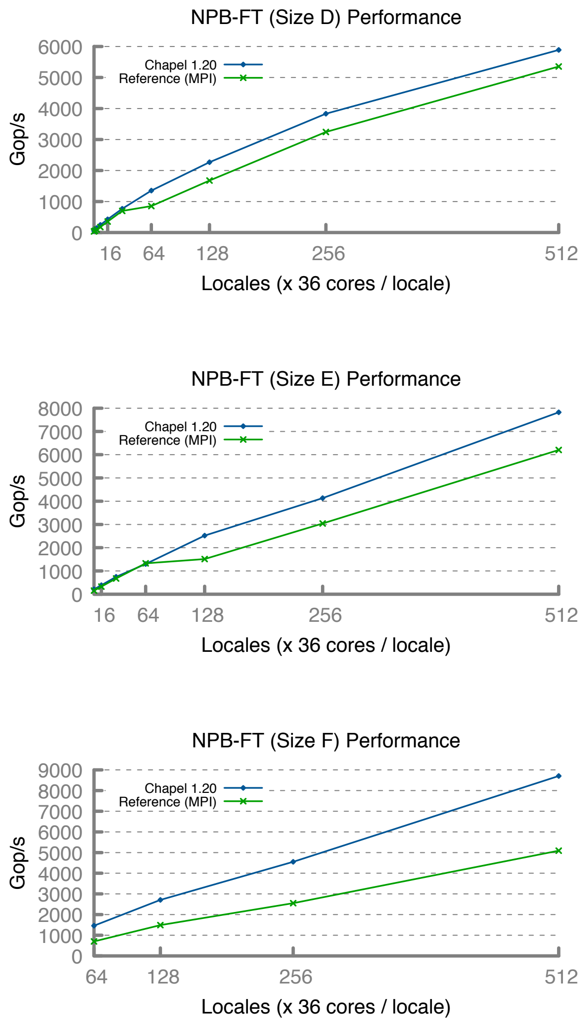

Fig. 1: Performance results for the NPB-FT benchmark (up is better).

REFERENCES

[1] L. Hui, J. P. Ostriker, S. Tremaine, and E. Witten, "Ultralight scalars as cosmological dark matter," *Phys.Rev.D*, vol. 95, no. 4, p. 043541, Feb 2017.

[2] J. Veltmaat and J. C. Niemeyer, "Cosmological particle-in-cell simulations with ultralight axion dark matter," *Phys.Rev.D*, vol. 94, no. 12, p. 123523, Dec 2016.

[3] H.-Y. Schive, J. A. ZuHone, N. J. Goldbaum, M. J. Turk, M. Gaspari, and C.-Y. Cheng, "GAMER-2: a GPU-accelerated adaptive mesh refinement code - accuracy, performance, and scalability," *Month.Not.Royal.Ast.Soc*, vol. 481, no. 4, pp. 4815–4840, Dec 2018.

[4] F. Edwards, E. Kendall, S. Hotchkiss, and R. Easther, "PyUltraLight: a pseudo-spectral solver for ultralight dark matter dynamics," *JCAP*, vol. 2018, no. 10, p. 027, Oct 2018.

[5] X. Li, L. Hui, and G. L. Bryan, "Numerical and perturbative computations of the fuzzy dark matter model," *Phys.Rev.D*, vol. 99, no. 6, p. 063509, Mar 2019.

[6] The numpy home page. [Online]. Available: https://www.numpy.org

[7] The numba home page. [Online]. Available: https://numba.pydata.org

[8] J. W. Eastwood and D. R. K. Brownrigg, "Remarks on the Solution of Poisson's Equation for Isolated Systems," *Journal of Computational Physics*, vol. 32, no. 1, pp. 24–38, Jul 1979.

[9] The fftw home page. [Online]. Available: http://www.fftw.org

[10] M. Pippig, "Pfft - an extension of fftw to massively parallel architectures," *SIAM J. Sci. Comput., 35:C213*, p. 2013.

[11] C. Bell, D. Bonachea, R. Nishtala, and K. A. Yelick, "Optimizing bandwidth limited problems using one-sided communication and overlap," in *20th International Parallel and Distributed Processing Symposium (IPDPS 2006), Proceedings, 25-29 April 2006, Rhodes Island, Greece*, 2006. [Online]. Available: https://doi.org/10.1109/IPDPS.2006.1639320

[12] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, "The nas parallel benchmarks—summary and preliminary results," in *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, ser. Supercomputing '91. New York, NY, USA: ACM, 1991, pp. 158–165. [Online]. Available: http://doi.acm.org/10.1145/125826.125925