WHITE PAPER

SGI® Unveiling the Early Universe: Optimizing Cosmology Workloads for Intel® Xeon Phi™ Coprocessors in an SGI® UV™ 2000 System

Abstract

This white paper documents the optimization of a 3D stencil code, “WALLS”, developed by the COSMOS supercomputing facility at the University of Cambridge, UK. We demonstrate that straightforward parallel tuning techniques can deliver significant performance improvements on both Intel® Xeon® processors and Intel® Xeon Phi™ coprocessors, while maintaining code readability and platform portability.

Our changes accelerate the execution of “WALLS” by factors of 9.3 and 30.1 on processors and coprocessors, respectively, and a single coprocessor is shown to outperform two processor sockets by a factor of 1.3. These results highlight the benefits of tuning software to effectively exploit parallel hardware and demonstrate the utility of Intel® Xeon Phi™ coprocessors for cosmology workloads.

TABLE OF CONTENTS

1.0 Introduction

The discovery of the Higgs particle at the Large Hadron Collider (LHC) confirms that the Universe has a complicated underlying structure with broken symmetries. A more familiar type of broken symmetry is seen in a ferromagnet with its magnetised domains; the boundaries between each of these misaligned regions have additional energy and are called domain walls. The same can happen in our Universe: Higgs-like fields in different regions become aligned in different directions with domain walls separating them (or indeed other defects, such as cosmic strings).

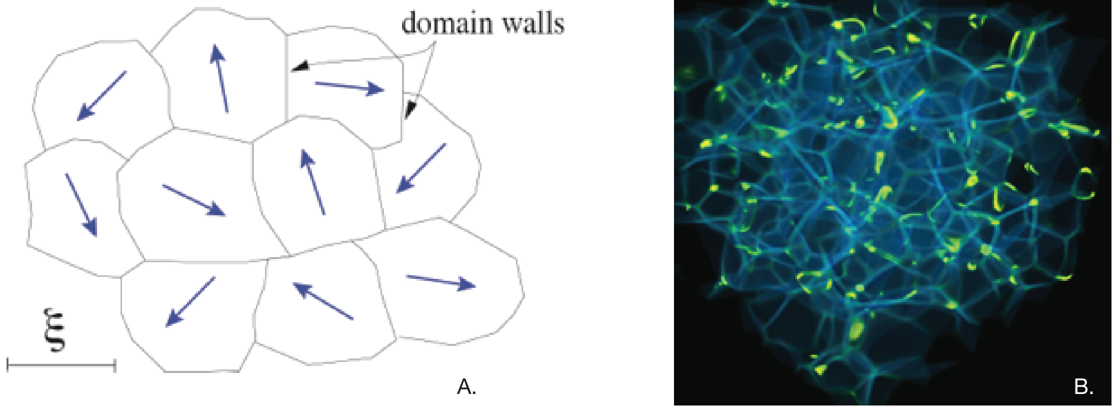

Figure 1: (a) Walls are generic phenomena wherever there are preferred directions in space, such as in a ferromagnet. (b) Cosmic walls can form complex hybrid networks stretching across the observed universe. Here, isocontours of the energy density are plotted, showing the lines along which walls intersect and overlap.

Walls have potentially important implications in the very early universe, as well as the late universe (i.e. the universe today). There are a number of observational motivations for this investigation. Domain walls can create distinctive imprints in the cosmic microwave background radiation because of their gravitational effects. One possibility is that low-energy walls stretching across the Universe could account for the apparent asymmetries observed by the Planck satellite. Like cosmic strings, they can also create signals that are similar to gravitational waves which were constrained recently using a joint analysis of the Planck and BICEP experiments.

The “WALLS” code simulates the evolution of a network of domain walls in the early universe (i.e. less than a nanosecond after the big bang), and has been developed into a number of variants -- complex hybrid networks with up to 100 different types of walls and strings have been investigated, as well as wall evolution in four and five space-time dimensions, with the latter inspired by fundamental theory. The reader is referred to the Stephen Hawking Centre for Theoretical Cosmology webpage [1] for more information.

In this white paper, we investigate the optimization and modernization of the simplest of the WALLS variants – a 3D implementation used as a benchmark for acceptance testing on new COSMOS supercomputers. A complete run of this benchmark configured for a 2048x2048x2048 problem requires 256 GB of memory, and takes approximately half an hour to complete 1020 iterations using more than a quarter of the COSMOS-IX supercomputer (512 Intel® Xeon® E5-4650L cores). There is a desire at COSMOS to run production simulations that are much larger than this, for more iterations, and preferably on fewer cores.

Although featuring simplified mathematics, the benchmark variant we use here remains representative in terms of its algorithmic and memory behaviors; the lessons we learn are easily transferable to the more complex cases, and are consequently of great value to COSMOS.

2.0 Implementation

"WALLS" creates random initial conditions on a 3D grid for a network of domain walls. It then solves dynamical field equations to find out how they evolve, computing the sum of the wall areas in every time-step.

Each time-step consists of three distinct algorithmic stages, using the equations below:

Benchmark System

1. Laplacian Stencil Operation

$$(\nabla^2\phi)_{ijk} \equiv \phi_{i+1,j,k} + \phi_{i-1,j,k} + \phi_{i,j+1,k} + \phi_{i,j-1,k} + \phi_{i,j,k+1} + \phi_{i,j,k-1} - 6\phi_{i,j,k}$$

2. Leap-frog Integration

$$\dot{\phi}_{ijk}^{n+1} = \phi_{ijk}^{n} + \Delta\eta\dot{\phi}_{ijk}^{n+1/2}, \text{ where } \dot{\phi}_{ijk}^{n+1/2} = \frac{\left[(1-\delta)\dot{\phi}_{ijk}^{n-1/2} + \Delta\eta\left(\nabla^2\phi_{ijk}^{n} - \frac{\partial V}{\partial\phi_{ijk}^{n}}\right)\right]}{(1+\delta)}$$

3. Area Calculation

$$\int n.dA = \Delta A \sum_{\text{links}} \delta_{-}^{+} \frac{|\nabla\phi|}{|\phi_x| + |\phi_y| + |\phi_z|}$$

```
#pragma omp parallel
for (int t = 0; t < nsteps; t++) {

    // Stage 1: Laplacian stencil using Phi at step t
    // Stage 2: Leap-frog integration, moving Phi to step t+1
    #pragma omp for
    for (int k = 0; k < Nz; k++) {
        for (int j = 0; j < Ny; j++) {
            for (int i = 0; i < Nx; i++) {
                …
            }
        }
    }

    // Stage 3: Calculate area of walls at step t+1
    #pragma omp for
    for (int k = 0; k < Nz; k++) {
        for (int j = 0; j < Ny; j++) {
            for (int i = 0; i < Nx; i++) {
                …
            }
        }
    }

}
```

Figure 2: Pseudo-code for "WALLS"

The first two algorithmic stages are implemented as a single loop over all of the cells in the grid, and the third stage as a separate loop (again over all of the cells in the grid). All iterations of the loops are known to be independent, and so both can be scheduled to be run in parallel using OpenMP (as shown in Figure 2).

The SGI® UV™ 2000 supercomputer currently deployed at COSMOS is a large symmetric multiprocessing (SMP) machine which, unlike a more typical cluster of interconnected nodes, presents itself to software as a single shared-memory system. The OpenMP parallelism in WALLS is therefore sufficient to enable the code to scale across sockets, cores and coprocessors without the use of a communication library such as MPI.

3.0 Experimental Setup

All experiments were performed on a single node of the COSMOS-IX supercomputer at the University of Cambridge, UK. Since "WALLS" is a stencil code that is expected to ultimately become memory bound, much of the optimization work can take place at this scale.

A summary of the hardware and software configuration is given in Table 1. Except where noted, we use all of the available cores on each platform and each core runs the maximum number of threads supported -- one per processor core (hyper-threading is disabled to improve machine stability) and four per coprocessor core. All comparisons between processor and coprocessor feature two processor sockets and a single coprocessor.

We use a 480x480x480 problem, to ensure that good data decompositions are possible on both platforms (i.e. 480 planes / 16 threads = 30 planes per thread; 480 planes / 240 threads = 2 planes per thread). Choosing a problem size that decomposes well on one platform but not on the other (e.g. 512 planes / 16 threads = 32 planes per thread; 512 planes / 240 threads = 2.133 planes per thread) would lead to load imbalance, and would give one platform an unfair advantage. It is possible to alleviate the effects of this by increasing the granularity of the exploitable parallelism (e.g. through loop restructuring or use of #pragma omp for collapse), but in choosing a "good" problem size here we are able to report and compare the best performance available on both platforms.

Table 1: Hardware and software configuration

	Intel® Xeon® E5-4650L Processor	Intel® Xeon® Phi 5110P Processor
Sockets x Cores x Threads	2 x 8 x 1	1 x 60 x 4
Clock (GHz)	2.60	1.05
Single Precision Peak (GFLOP/s)	665 (8 adds + 8 multiplies per cycle)	2021 (16 fused multiply-adds per cycle)
L1 / L2 / L3 Cache (KB)	32 / 256 / 20,480	32 / 512
DRAM (GB)	64	8
Max Bandwidth (GB/s)	51.2	320
Compiler Version	-icc 14.0.0 20130728	
OpenMP Environment	-KMP_AFFINITY=compact,granularity=fine	

4.0 Analysis

Out of the box, the original “WALLS” code runs 2x faster on two processors than it does on a single coprocessor. This is not in line with our expectations – typically, a single coprocessor can outperform two sockets by a factor of around 1.5 – suggesting that there are likely to be some opportunities to improve performance.

Performing a “General Exploration” analysis in Intel® VTune™ Amplifier, we see that the number of cycles per instruction (CPI) is very high and that a large number of L1 misses are not being serviced by L2 cache. Additionally, examining the compiler’s vector report output (from –vec-report), we see that the main compute loops do not vectorize. The results of this initial high-level analysis explain the difference in runtime we see between the two platforms, since much of the performance of Intel® Xeon Phi™ coprocessors comes from their increased memory bandwidth and wider vector units (relative to Intel® Xeon® processors).

In the following sections, we explore several changes to “WALLS” designed to address the performance issues we have observed. Broadly speaking, these changes can be categorised as either optimizations (i.e. code-level changes that improve performance on a fixed hardware platform) or modernizations (i.e. algorithmic changes that are likely to be of benefit on any modern architecture). For example, under this categorization system, better utilization of a particular instruction set would be considered an optimization; increasing the amount of exploitable parallelism, or improving the flop:byte ratio, would be considered a modernization.

5.0 Optimization and Modernization

5.1 Enabling Auto-vectorization

Although this white paper focuses on the performance of 3D simulations (the most typical use case at COSMOS today), the version of “WALLS” that we use supports problems of up to four dimensions – as a result, an NxNxN problem is actually handled by default as an NxNxNx1 problem. The fourth dimension is traversed in the code’s inner-most loop and, since this loop only ever performs a single iteration, auto-vectorization should not be expected to provide any speed-up. Our first change to the code simply alters the loop order, such that the problem is treated as 1xNxNxN.

Compiling with –vec-report3 and –vec-report5 causes the compiler to output diagnostic messages describing the reason that each loop in the program could not be vectorized, along with a list of any assumed inter-iteration dependencies. The most common of these assumed dependencies in C programs are caused by the use of pointers – for correctness, the compiler must assume that two pointers could point to the same memory. Marking pointers with the restrict keyword or adding #pragma ivdep immediately before a loop allows the compiler to ignore its assumptions.

In “WALLS” we see another kind of dependency, where a variable is used to accumulate the sum total of all wall areas. In the original code, this accumulation step is guarded by a branch (which checks whether a wall exists), and the compiler therefore does not identify this sum as a reduction. We can fix this either by moving the addition outside of the branch (adding zero when no wall exists), or by declaring the reduction explicitly (using #pragma simd reduction(+:sum).

We also encounter a less common diagnostic message: “remark: loop was not vectorized: operator unsuited for vectorization”. The operator in question is modulo (%) which “WALLS” uses to handle its periodic boundary condition – when addressing neighbour cells during the stencil operation, the next i index is computed as (i+1) % Nx, and the previous index as (i-1 + Nx) % Nx. Intel® Xeon Phi™ coprocessors do not have a packed modulo instruction, which prevents loops containing modulo operators from being vectorized.

An alternative way of handling this periodic boundary condition – one that the compiler is able to vectorize – is to replace the % operations with simple branches via C's "ternary operator", as below:

```
int im1 = (i > 0) ? i-1 : Nx-1;
int ip1 = (i < Nx-1) ? i+1 : 0;
```

The compiler can generate vector masks and masked move instructions for this code, enabling it to be effectively vectorized.

5.2 Improving Auto-vectorization and Cache Behavior

Although "WALLS" auto-vectorizes after applying the code changes detailed in the previous section, handling boundary conditions in this way is not efficient. Even though the vast majority of the loads encountered during our stencil will be contiguous, the compiler may choose to use a sequence of relatively expensive gather/scatter operations instead of simple packed loads (due both to the presence of the branch, and to border cells requiring data from the other "end" of the array).

There are two alternative optimizations that we could employ to handle the periodic boundary condition more efficiently: 1) peeling the first and last iterations from the inner-most loop, such that all of the remaining iterations are known not to handle any edge cases; or 2) introducing "halo" or "ghost" cells (i.e. a layer of additional cells around the grid which store values representative of the boundary condition).

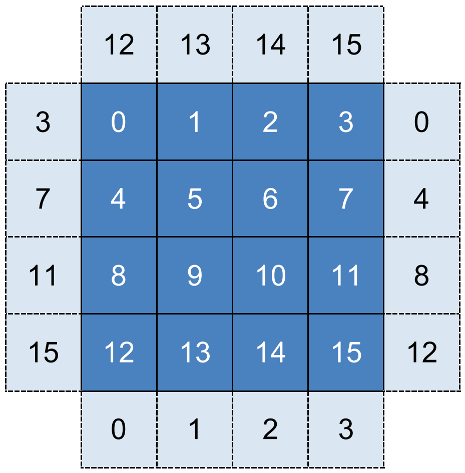

Figure 3: A 4x4 grid with halo data, for a periodic boundary condition

Such halo cells are a commonly used design pattern in many high performance applications, frequently appearing alongside domain decompositions. For a periodic boundary condition, each halo cell must store an up-to-date copy of the value on the opposite side of the volume (as shown in Figure 3), and thus we must introduce additional code to copy data from the grid into the appropriate halo cells between time-steps.

The memory overhead of such a halo scheme is negligible for sufficiently large simulations. For example, adding halo cells to an array of 1024x1024x1024 doubles increases its 8 GB footprint by only 48 MB.

The use of halo cells in this manner doesn't only improve vectorization, but also cache behavior – a cell is much "closer" in memory to a neighbouring halo cell than to a cell on the other side of the array. Indeed, introducing a halo only in the X dimension is sufficient to improve vectorization (since taking a step in the Y or Z directions will always give a contiguous run of X values), but we see benefits from introducing halos in all three dimensions.

```
#pragma omp for
for (int k = 0; k < Nz; k++) {
    for (int j = 0; j < Ny; j++) {
        im1 = ni; ip1 = 1;
        …
        for (int i = 1; i < Nx-1; i++) {
            im1 = i-1; ip1 = i+1;
            …
        }
        im1 = i-1; ip1 = 0;
        …
    }
}
```

A.

```
#pragma omp for
for (int k = 0; k < Nz; k++) {
    for (int j = 0; j < Ny; j++) {
        for (int i = 1; i < Nx+1; i++) {
            im1 = i-1; ip1 = i+1;
            …
        }
    }
}
```

B.

Figure 4: Preparing 3D loop for vectorization using (a) loop peeling; and (b) halo cells.

Additionally, using halo regions results in code that is much simpler and consequently more readable and maintainable than a loop with peeling (as demonstrated by Figure 4, in which all ellipses represent an identical loop body).

5.3 Avoiding Slow Vector Operations

It is well known that some arithmetic operations run faster (i.e. in fewer clock cycles) than others. Floating-point division operations specifically are significantly slower (in terms of latency and throughput) than a floating-point add or multiply, and are a performance bottleneck in many scientific applications.

A common optimization is to replace divisions with multiplications by reciprocals, and the Intel® Xeon Phi™ coprocessor features hardware support for approximate reciprocals (the accuracy of which can be improved through Newton-Raphson iteration) to accelerate the cases where this transformation is permissible. When the division is by a constant, even greater speedups are possible, since the reciprocal of that constant can be pre-computed and reused in multiple calculations.

It is this last case (i.e. division by a constant) that appears most commonly in "WALLS"; for example, we can see that the calculation of $\dot{\phi}_{ijk}^{n+1/2}$ can be rewritten as:

$$\dot{\phi}_{ijk}^{n+1/2} = \left[(1-\delta)\dot{\phi}_{ijk}^{n-1/2} + \Delta\eta\left(\nabla^2\phi_{ijk}^{n} - \frac{\partial V}{\partial \phi_{ijk}^{n}}\right)\right] \times \frac{1}{1+\delta}$$

where δ is a constant. We can therefore reuse a pre-computed reciprocal for every cell in every time-step, replacing hundreds of thousands of division operations with multiplications.

Enabling certain compiler options (e.g. -no-prec-div, -fast) may be sufficient to allow the compiler to perform this and similar optimizations without any source modification. For "WALLS", we found that this was not the case – although the compiler was able to replace the divisions by reciprocals, it did not identify that they could be hoisted outside of the loop.

5.4 Reducing Memory Footprint

Although the complete COSMOS-IX system has a large amount of memory available, it is important that we do not waste memory unnecessarily. Efficient memory utilization is especially important when looking to use a coprocessor, since they have a limited amount of RAM (8 GB).

The code's original implementation uses a significant amount of memory -- four 3D arrays of doubles storing $\phi_{ijk}^{n}, \phi_{ijk}^{n+1}, \dot{\phi}_{ijk}^{n+1/2}$ and $\dot{\phi}_{ijk}^{n-1/2}$.

Rearranging $\phi_{ijk}^{n+1} = \phi_{ijk}^{n} + \Delta\eta\dot{\phi}_{ijk}^{n+1/2}$ as $\dot{\phi}_{ijk}^{n+1/2} = [\phi_{ijk}^{n+1} - \phi_{ijk}^{n}]/\Delta\eta$, we see that it is possible to compute half-time-step quantities from full-time-step quantities, and thereby decrease the code's memory footprint by a factor of two. The decrease in the number of arrays we must stream through in a given time-step also decreases our memory bandwidth requirements, and places less pressure on prefetching and paging hardware.

5.5 Specializing Code for Problem Dimensionality

As mentioned previously, the original "WALLS" code treats all 3D problems as special cases of 4D problems. This introduces a significant amount of wasted work -- for example, the Laplacian stencil operator in Stage 1 is coded as:

```
phi = P[l-1][k][j][i] + P[l+1][k][j][i] +
      P[l][k-1][j][i] + P[l][k+1][j][i] +
      P[l][k][j-1][i] + P[l][k][j+1][i] +
      P[l][k][j][i-1] + P[l][k][j][i+1] + …
      - 8 * P[l][k][l][i];
```

where every cell's neighbour in the fourth (l) dimension is itself, due to the periodic boundary condition. The code therefore performs two unnecessary loads and additions per grid cell in every time-step. The problem dimensionality (and problem size) used by "WALLS" are not decided at runtime but rather at compile time, and we are therefore able to remove these unnecessary calculations by specializing code generation for the 3D and 4D cases through the use of #if guards (as shown in Figure 5).

```
#if DIMENSION == 3
laplace = 6;
#elif DIMENSION == 4
laplace = 8;
#endif
…
phi = P[l][k-1][j][i] + P[l][k+1][j][i] + … ;
#if DIMENSION == 4
phi += P[l-1][k][j][i] + P[l+1][k][j][i];
#endif
phi -= laplace * P[l][k][l][i];
```

Figure 5: Generating code specialized for 3D and 4D stencil operations.

The importance of performing such specialization is increased by our earlier decision to introduce halo cells, since the dimensionality of the halo data required changes with the dimensionality of the problem: individual cells in 1D; rows and columns in 2D; planes in 3D; and cuboids in 4D. The halo data in the fourth dimension for a 1xNxNxN problem is actually two complete copies of the entire NxNxN grid; therefore, specialization does not only avoid wasted work but wasted memory as well.

5.6 Fusion of Algorithmic Stages

As shown in Figure 2, the two loops in "WALLS" are necessary because the algorithmic stages operate on different time-steps: the stencil is computed for step t, the simulation advances to step t+1 and the wall areas are computed for step t+1.

It is possible to combine all three algorithmic stages into a single loop through some simple restructuring of the computation – specifically, by computing the wall areas for step t at the same time as computing the stencil. "Fusing" the two loops in this manner improves memory behavior significantly, since we need only stream through the phi array once per time-step instead of twice.

We must introduce a prologue (i.e. a single time-step without any area calculation) in order to mimic the application's original behavior, but the overhead of this prologue is amortized across the remaining time-steps.

6.0 Results and Conclusions

The graph in Figure 6 shows the speedups (higher is better) resulting from each of our optimizations and modernizations. We calculate speedup relative to the performance of the original code running on two Intel® Xeon® processors; therefore, the tallest bar represents not only the greatest speedup, but also the best overall runtime. Note that we treat the introduction of halo cells in the inner-most and outer loops as different optimizations (Halo 1D and 3D, respectively) to separate the speedups arising as a result of improvements in vectorization and cache behavior.

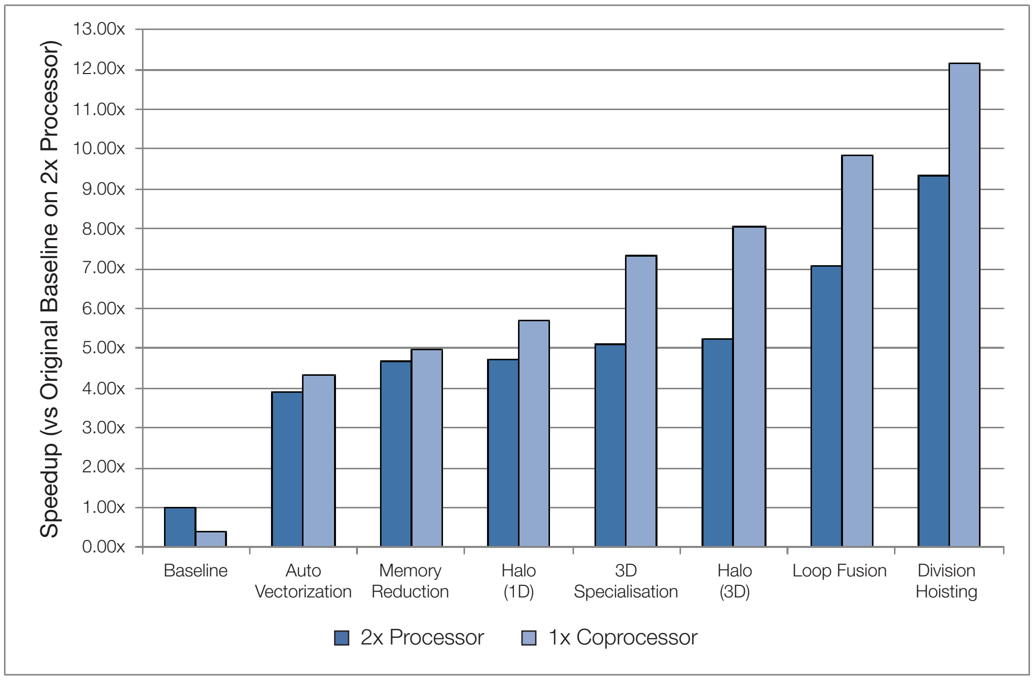

Figure 6: Achieved speedup for each of our optimizations and modernizations, relative to the original code running on two Intel® Xeon® processors.

There are three important takeaways from these results:

1. For this workload, enabling auto-vectorization alone is sufficient to make the coprocessor outperform two processors. Vectorization improves processor performance by 3.9x and improves coprocessor performance by 10.8x. The coprocessor speedup is greater than the vector width because we also avoid expensive scalar operations (e.g. modulo).
2. Each of our optimizations and modernizations benefit both processor and coprocessor. That the coprocessor benefits more is due to its increased sensitivity to vectorization and memory behavior.
3. The speedups we see on each platform are large enough to have significant impact upon scientific discovery at COSMOS. The final version of the code runs 9.3x and 30.1x faster than the original code running on processors and coprocessors respectively, and a single coprocessor outperforms two processors by 1.3x.

This white paper therefore demonstrates that it is possible to deliver dramatic performance improvements to real-life codes through straightforward changes that do not impact platform portability, maintainability or code readability.

6.1 Future Work

The current version of the code computes the wall areas (and a number of other diagnostic values) in each time step, performs a reduction operation across the threads and then, for certain timesteps, writes these values to file. The cost of synchronization at this point is higher on the coprocessor (due to the increased number of threads), and subsequently harms performance. We are in the process of investigating several methods to reduce this synchronization cost, including: computing the diagnostic values less frequently (e.g. every 10 timesteps); and computing the diagnostic values at the same frequency, but reducing them less often.

The optimizations and modernizations explored here have focused on improving single node performance. The code's ability to scale to larger problems running on hundreds of nodes is a key requirement – after a sufficient number of time-steps, the presence of a boundary condition (which does not exist in the real universe) begins to have an effect on wall evolution, limiting the amount of real-world time that can be simulated using a fixed memory footprint.

The current generation of Intel® Xeon Phi™ coprocessors cannot access the full memory available in COSMOS-IX, limiting native runs of "WALLS" to simulations that fit in 8 GB of memory. We are currently investigating the use of the pragma-based Intel® Language Extensions for Offload (LEO), and the similar functionality available in OpenMP* 4.0, as a means of overcoming this limitation.

7.0 Additional Resources

[1] The Stephen Hawking Centre for Theoretical Cosmology, www.ctc.cam.ac.uk/outreach/origins/cosmic_structures_two.php

[2] W.H. Press, B.S. Ryden and D.N. Spergel, "Dynamical Evolution of Domain Walls in an Expanding Universe", Astrophys. J. 347 (1989)

[3] A.M.M. Leite and C.J.A.P Martins, "Scaling Properties of Domain Wall Networks", Physical Review D 84 (2011)

[4] A.M.M. Leite, C.J.A.P Martins and E.P.S. Shellard, "Accurate Calibration of the Velocity-Dependent One-Scale Model for Domain Walls", Physics Letters B 7 18 (2013)

8.0 About the Authors

Dr. John Pennycook is an Application Engineer at Intel.

James Briggs is COSMOS Parallel Programmer in the Department of Applied Mathematics and Theoretical Physics, University of Cambridge (UK).

Prof. Paul Shellard is a member of the Department of Applied Mathematics and Theoretical Physics, University of Cambridge (UK).

Dr Carlos Martins is a Senior Research Fellow at the Centre for Astrophysics, University of Porto.

Michael Woodacre is a Chief Engineer at SGI.

Karl Feind is a Principle Engineer at SGI.

9.0 About COSMOS

The COSMOS consortium are among Europe's leading cosmology researchers, a team brought together by Prof. Stephen Hawking in 1997. There is participation from cosmology groups in ten UK Universities with interests ranging from inflation in the very early universe through to science exploitation of the Planck satellite survey of the cosmic microwave sky. The COSMOS supercomputer is part of the UK Distributed Research utilising Advanced Computing facility (DiRAC); it is housed and operated on behalf of DiRAC by the Centre for Theoretical Cosmology within the University of Cambridge.

COSMOS-IX is an SGI® UV™ 2000, with 1864 Intel® Xeon® processor cores, 31 Intel® Xeon Phi™ coprocessors and 14.8 TB of globally shared memory. The highly scalable SGI® UV™ 2000 architecture has the unique ability to support up to 32 Intel® Xeon Phi™ coprocessors within a single Linux image, which enables threaded, OpenMP, MPI, and OpenSHMEM applications to scale up problem size, as well as the number of utilized processors and coprocessors.

This shared-memory environment enables fast prototyping and the rapid development of complex data analysis pipelines in a highly competitive and fast-moving field. Combined with Intel® Xeon Phi™ coprocessors, the platform supports the widest range of parallel programming paradigms available, including offload acceleration, while supporting heterogeneous workflows with different components best suited to specific processor technologies.

10.0 About SGI

SGI is a global leader in high performance solutions for compute, data analytics and data management that enable customers to accelerate time to discovery, innovation, and profitability. Visit sgi.com for more information.

Global Sales and Support: sgi.com