6.338 project report: Pattern-Forming Instabilities in the Swift-Hohenberg Model Micah Brodsky

### Problem overview

This project considers the efficient parallel implementation of a time-domain solver for the Swift-Hohenberg family of pattern-forming partial differential equations, to facilitate hands-on exploration of its parameter space and the behavior of high-amplitude patterns and perturbations. Swift-Hohenberg is commonly used as a "model equation" for studying pattern-formation phenomenon, as it is one of the simplest possible pattern forming systems and it can be derived as a low-amplitude approximation to certain physical phenomena such as Bénard convection. While a variety of its phenomena can be investigated analytically using perturbational techniques, many subtle features and high-amplitude behaviors are difficult to demonstrate or difficult to anticipate. Thus, a numerical solver is a valuable investigative tool.

The basic, dimensionless form of the Swift-Hohenberg equation is

### $\partial u / \partial t = \varepsilon u - (\nabla^2 - 1)^2 u + g_2 u^2 - u^3 + higher order nonlinearities$

The linear portion of this equation is unstable and frequency-selective (peak gain at k = 1), providing the basic template for pattern formation.  $\varepsilon$  is the "supercriticality" parameter, determining the magnitude of the instability. The cubic term, however, eventually causes saturation and thus keeps the system from blowing up. The  $\varepsilon$  and nonlinear terms collectively determine the character of the resulting pattern, by controlling the stability and frequency flow of the system. As posed, the system is a "gradient equation", monotonically following the gradient of its Lyapunov functional downhill towards a steady state pattern.

This project considers the equation within a 2D rectangular domain, using either fixed or periodic boundary conditions. (No special attention is paid to the effects of boundary conditions on the resulting patterns, although such effects are sometimes subtle and nontrivial.) Initial conditions can be either random or programmatically structured. Results in the form of periodic snapshots of the domain are animated live as well as written to a file for later analysis.

## **Distinguishing features**

At a high level, the problem at hand is similar to other finite difference time domain PDE integrators for simple domains. Explicit methods are easy to implement and straightforward to parallelize, using slab or block decomposition and data exchange with nearest neighbors each iteration. In memory access, spatial locality is extremely high but temporal locality is weak, putting pressure on memory bandwidth. Besides computational efficiency, time step is an important consideration in performance.

However, several differences change the computational characteristics of this problem and make it distinctly harder to extract good serial performance, as compared to simpler problems like the diffusion and wave equations. The most obvious difference is the nonlinearity, which makes implicit and spectral

methods even more difficult. But, the unusually high spatial order  $\nabla^4$  in the equation makes the explicit methods used highly unstable, necessitating a very small time step.

Another important difference is the complexity of the "kernel" applied at each cell at each time step. Using the 13-cell, 5x5 stencil implied by the straightforward finite-difference formulation of the equation requires 13 memory loads and roughly 30 FLOPs. The diffusion equation, in contrast, requires 5 memory loads and about 6 FLOPs, and the memory loads are easily shared across cells with loop-unrolling. This nearly fivefold relative increase in kernel complexity makes local computation a serious bottleneck, diminishing the importance of memory and communication bandwidth.

The target architecture, the SiCortex SC648, also has an unusually high ratio of communication bandwidth to serial processing speed. Each SiCortex processor node operates at only 500MHz. Clock-for-clock, the individual nodes are even slower, as they have very limited ability to exploit instruction-level parallelism compared to modern desktop processors. The machine's peak link bandwidth of 2GBps, however, is an order of magnitude faster than common cluster interconnects such as gigabit Ethernet. As a result, each node can transfer as much as one double per two clock cycles; communication overhead becomes surprisingly small relative to compute time. Given the large number of processors per machine, this unique design point makes good serial performance very difficult but scalable parallelism quite easy. Combined with the high kernel complexity of the problem at hand, scaling is almost perfect, the computational load completely outpacing communication overhead.

However, the high kernel complexity combined with the slow time step also leads to another qualitative change: While many PDE problems require parallelization because of the sheer size of their domains, this problem requires parallelization merely because of the high computational load. For typical problem sizes, the entire per-processor slice is smaller than the L2 cache. Because of the kernel complexity, L2 cache effects are relatively minor, but the small domain also presents a serious problem for scalability. With slab decomposition, an n-width domain can only be straightforwardly split over n processors (n/2, in fact, without special considerations and additional overhead). In most runs, the width is only several hundred cells. On the SiCortex machine, with numerous but slow processors, this, not overhead, provides the ultimate limit to scalability.

## Implementation details

The project is implemented in C++ using MPI. The domain is represented as a column-major 2D array of doubles, decomposed into per-processor slabs along the horizontal dimension. Initial conditions are established programmatically, and then the system is iterated using explicit finite differences. Two buffers are maintained, an input buffer and an output buffer, and the pointers are swapped with each iteration. The original, un-optimized version of the code traverses by columns, moving from left to right, while the optimized version follows a hybrid strategy (described below). Between iterations, processors exchange 2-cell-wide border regions with their immediate neighbors, using MPI\_Sendrecv in the original version and non-blocking sends in the optimized version. At periodic intervals, the processors perform a gather operation to copy the entire domain to the head node, which optionally plots the data on a live display using the Simple DirectMedia Layer (SDL) library via X11 tunneling or writes it to a dump file.

The figure below shows at a high level the performance characteristics of the code for a large domain size. Both the original and the optimized versions are shown. The figure plots iterations per second vs. number of processors for a 1200x800 domain. Scaling is nearly perfect up to 600 processors, where it abruptly stops. This is a slightly wide domain, in order to illustrate the performance scaling properties of the code all the way to the full capacity of the host machine. More common domain sizes would cease to scale around 300 processors, at which point the domain cannot be further subdivided because the slabs have the minimum width supported by the code, two cells.



# Optimizations

This section describes the optimizations and refinements that improved performance by 20%-40% over the original version of the code. The improvements came in small increments and proved surprisingly difficult to achieve; gains in one area frequently competed with associated losses in another. Kernel complexity and instruction count, above all, seemed to impose an upper bound on computational performance improvements.

### Overlapped communication & computation

The largest obvious target for optimization is communication overhead. It peaks at just shy of 40% of runtime (e.g. 1200-width domain on 600 processors, profiled using mpipex), limited by the SiCortex machine's high bandwidth and the kernel's high complexity, but still quite significant. Most of this time could, in principle, be eliminated. One possible strategy would be to use a two-dimensional domain decomposition with thicker, shorter slices, increasing the ratio of interior area to perimeter. This comes at the expense of potentially higher overhead and greater latency due to four-neighbor communication, however. Another strategy would be to use non-blocking communications, overlapping communication with computation. Given that computation time exceeds communication time, one might expect that non-blocking communication should be able to extract most of the available benefit. In practice, the improvement was significant -- up to 25% in the extreme case -- but not nearly as good as the theoretical "Amdahl" limit.

Nonblocking communication (using MPI\_Isend and MPI\_Irecv) was implemented by subdividing the perprocessor domain slice into three components: left edge, right edge, and body. Computation on the left edge is performed first. As soon as this is complete, the message to the leftward neighbor can be sent. The right edge and the transmission to the right neighbor follow. Then, the body is processed. This allows each edge transmission to continue in the background while the other edge and the body are processed.

In practice, this fails to completely eliminate communication overhead, because the case when communication overhead is most significant is precisely when the body component is thin or absent. This limits the time allowed for background message transmission to as little as half the total compute time. Messages frequently fail to arrive in time for slab widths of about 6 or smaller, requiring synchronous waits. It should be possible to significantly reduce this problem by breaking the edge components up into

slices and sending each slice as a separate message as soon as it is ready. However, this would introduce substantially more MPI overhead, and it is not obvious whether a net improvement would result. Alternatively, a 2D block decomposition would reduce the size of the messages and the fraction of computation in the edge components, which might eliminate this slowdown.

Another, more subtle issue limits the effectiveness of overlapped communication: Because the edges must be computed separately and before the body, they cannot fully participate in the L1 cache locality optimizations described below, adding some unexpected overhead to the non-blocking code. This issue might also be reduced by using 2D block decomposition.

L1 enhancements and "tiling"

The wide  $\nabla^4$  stencil's repetitive access to old columns suggests an opportunity for locality optimization. For "tall" domains where the memory size of a column approaches the size of the L1 cache, previously traversed columns are likely to be evicted from the cache before the stencil has finished with them. Indeed, profiling with papiex -a shows a noticeable drop in L1 cache hit rate, from about 97% to 93%, as height grows from 200 to 800. The resulting added 9% stall time is perhaps surprisingly small, the cache misses seemingly drowned out by the high instruction count of the kernel (including many loads with high spatial locality), but it does represent room for improvement.

Nearly all of this lost time can be recovered by changing the order in which the domain is traversed. Traversing by row would not solve the problem (and would destroy spatial locality), but a hybrid strategy, dividing the domain up into blocks of rows and traversing each block by column, does. (See figure below.)



To explore this and related strategies, the inner loop of the code was refactored into a "tiled" version -- a function that took x, y, width, and height parameters and ran the iteration over the specified (width, height) tile. This was then invoked from an outer loop iterating over 128-row slabs. (These tile routines were also used to cleanly implement the left, right, middle decomposition used in the previous section.) The resulting code brings the L1 hit rate back to about 97% for height 800 and improves performance by about 8%.

Additional, small gains for a narrow range of problem sizes might be possible by reversing traversal directions with each iteration, so that the most recently processed cells, likely still cached, are the first to be processed with the next iteration. This has not been explored.

### Compiler tricks

As is commonly the case, the compiler exerts a significant impact in the overall performance of this code. This project used the PathScale C++ compiler, a highly aggressive, GCC-compatible optimizing compiler, which ships with the SiCortex machine. For reasons not yet explored, it produces code that is a full five times faster than GCC -O3! However, one particular refinement made a noticeable improvement the code output of the PathScale compiler. In the original code, the compiler makes poor reuse of registers, register-cached values, and software pipelining. It appears that the use of buffer swapping in the iteration code was not successfully deciphered by the pointer aliasing analysis, forcing the compiler to conservatively assume that the two buffers could possibly point to the same memory. By declaring the buffer pointers

<u>\_\_restrict\_</u>, the compiler is informed that the pointers are in fact unique references to their memory targets. For simpler kernels, this resulted in as much as a 50% speed-up. For the more complicated kernel at hand, however, it provides about 10%.

#### Pre-computed Laplacian

An apparent opportunity for optimization exists in explicit 5x5 stencil formula used for the  $\nabla^4 u$  term. Besides its "wide" memory access pattern discussed above, it also is computationally expensive, requiring about 16 FLOPs and several index calculations. There is a common sub-expression hidden inside, however:

the finite-difference  $\nabla^4 u$  is simply the finite-difference Laplacian of the finite-difference Laplacian. Separately computing the Laplacian (also needed) into a buffer and then computing the repeated Laplacian

of that buffer for the  $\nabla^4$  term replaces a complex, 16-FLOP calculation with a simpler, 6-FLOP calculation, cutting the total number of FLOPs in the kernel by about 1/3. Shouldn't one expect this to improve performance?

Unfortunately, the result is quite the opposite. The trivial implementation, using a Laplacian buffer the same size as the local data buffers, results in slowdowns on the order of 10%. The cause appears to be worsened memory performance. Assuming the cache optimizations above eliminate L1 capacity misses due to reading from adjacent rows, the original implementation requires two compulsory L1 misses per cell (one for the input buffer, one for the output buffer). The trivially modified version here requires *five*: one for the input buffer while computing the Laplacian, one for the Laplacian buffer as output, then again one for the input buffer while computing the iteration, one for the Laplacian buffer as input, and one for the output buffer. With the SiCortex's 12-cycle L1 miss penalty, this is a waste of 36 cycles per cell per iteration -- significantly more than the cost of 10 FLOPs and their associated operand fetches.

With some cleverness, the number of compulsory misses can be reduced to three, by computing the Laplacian buffer continuously as main kernel iterates through the cells, at the cost of some overhead. This *almost* brings the speed up to break-even. Compulsory misses can even be reduced back to two by using a small, rolling buffer in the continuous computation. However, the overhead of managing such a buffer seriously cuts into potential gains. No performance improvement was ever demonstrated.

### Next steps

What are the interesting possible next steps for this project? Among computational improvements, 2D block decomposition is very attractive -- not because it would reduce communication overhead, but because it would push out the scalability limit for this small-domain, computationally-limited problem. However, the greatest possible improvements would seem to lie with alternative numerics, so that the tiny time step the explicit method requires could be enlarged. This is not an obvious win. Implicit methods would require solving 13-diagonal linear systems in parallel, which is likely to be far more expensive per iteration than explicit methods. Improvements may only result if the time step can be improved by orders of magnitude. Spectral methods might be a promising alternative, but the trivial FFT-based approach would require repeated, parallelized forward and reverse transforms in order to compute the nonlinear effects without quadratic overhead. More sophisticated spectral techniques would likely be necessary.

With either alternative solution method, the parallel interoperability problem would also come to the fore. An implicit method would require interfacing with a parallel linear algebra package such as ScaLAPACK. Alternatively, spectral methods would likely involve interfacing with transform libraries such as FFTW. In both cases, the wide acceptance of the MPI platform and the communicator abstraction makes direct interoperability conceivable, but preparing and distributing the data as expected by the libraries is distinctly nontrivial. Parallel FFTW, for example, dictates its own distribution of columns per node, rather than interrogating the client code. 2D block distributions are not even supported. It remains to be seen how challenging and how expensive communicating with these libraries in the inner loop would be.