# Overview

<https://en.wikipedia.org/wiki/Amdahl%27s_law> describes the speedup limits caused by serial portions of parallel algorithms.

Simply put, if a serial program executes for S + P time, and you manage to make P get spread over T threads, the new execution time is S + (P+L)/T where L is the additional time caused by locking shared resources.

But in our case, the situation is more complex. We have two choices for parallel execution – Xeon and Xeon Phi. Xeon Phi has more cores, no L3 cache, a faster L4 HBMW cache. It has similar vector performance to Xeon per VPU. But it has slower serial execution.

So we need to rewrite the serial program’s execution time formula as S + Pscalar + Pvector

Then, ignoring locking, the

Xeon execution is approximately S + Pscalar/20 + Pvector/20 20 cores

Xeon Phi execution is approximately S\*4 + Pscalar\*4/60 + Pvector/60 60 cores, each ¼ the speed of a Xeon core

For Xeon Phi to win, S\*0.75 - Pscalar\*0.017 + Pvector/30 > 0

i.e S\*0.75 > Pscalar\*0.017 – Pvector/30

S > Pscalar\*0.02 – Pvector/22

But we can pretend S is zero, and get Pvector > Pscalar\*0.44

It is not clear that we have meet this goal – more measurement is needed – but the measurements will not be valid until the following concern is addressed.

However this coarse calculation assumed L to be zero. L is not zero. L is larger on Xeon Phi than it is on Xeon because there are more cores and the cores are slower.

What are the causes of L – the serialization due to locking?

# Causes of Lock Serialization

In our case there are four causes of serialization

1. The serialization by locks within the parallel portion of the code when accessing the heap
2. The serialization by locks within the parallel portion of the code when accessing shared data
3. The serialization by the shared memory subsystem caused by missing the private caches

The code should avoid the heap as much as possible, and avoid shared data as much as possible.

The only way to avoid locking in the memory subsystem is to require make sure the memory traffic is somewhat lower than the max memory bandwidth.

# Our Memory Bandwidth Needs

The Xeon Phi can do approximately 6T Flops/sec, and each Flop needs about 8 bytes of data from the memory subsystem, so about 60TB/sec

The HBWM can supply about 600GB/sec – 1/100th of this. The DRAM about 60GB/sec - 1/1000th of it.

That means each number fetched from the HBWM must feed about 100 instructions, but our current algorithms don’t have this happening. Instead a typical fetch it is feeding between 1 to 5 instructions before being evicted from the L2 caches.

This observation is true for both Xeon and Xeon Phi. We need to increase our L2 hit rates from 99.9% to 99.999%, and can afford to do 10x more floating point operations to achieve this goal.

The main reason our usage rates are low is because our algorithms typically implement f(g(image)) by computing and storing g(image) and then later apply h to the stored g.

# Reducing Memory Traffic

The simple solution therefore is to NOT compute and store g(image), but to delay computing it until it is needed, and to recomputed it if it is needed again much later – only storing if the L2 cache is big enough.

We have several operations that we do on images – scale, rotate, translate, mask. Almost all of these are memory bound – only do a few floating point ops per cell.

Almost all can be easily expressed in a manner that will support delayed evaluation of cacheable sub-matrices. It should be easy to code in a way that can be easily flipped between either delaying and combining ops, or doing the op immediately, without the main algorithm code having to change.

# Coding Implications

To code this cleanly will require using C++ and the optimizing compilers expertly, but it should not mess up the expression of the algorithm itself.

I am beginning to explore this with some code prototypes.