1 Matrix Multiply

1.1 Scaling Plot

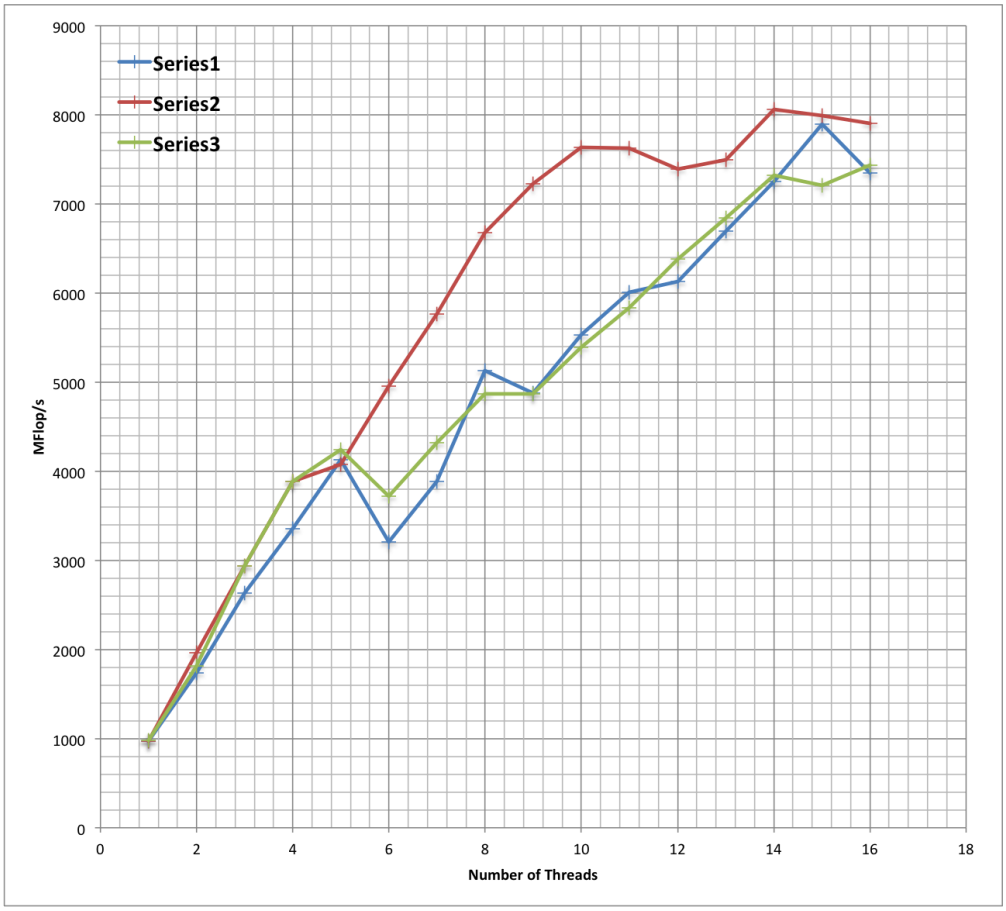

Figure 1: series 1: omp for; series 2: omp task; series 3: pThreads

1.2 Peak Performance

I changed the for loop ordering for all matrix multiply implementation to make the process a little faster. So, the scalar baseline was $970 MFlop/s$ and the best parallel performance was $8060 MFlop/s$ with 14 threads using the OpenMP task implementation. Hence, the speed-up was 8.3 time faster than the baseline.

No, we are not achieving linear speed-up. The speed-up is about 0.86 performance improvement per thread for the first ten threads. For the runs of 1 thread through 10 threads the performance increases linearly. However, it is not an ideal linear scaling as the speed-up is less then 1 performance improvement per thread. And also the performance starts flattening out by running the program on 11 through 16 threads.

The OpenMP Task implementation is the one that performs the best because it deals better with load balancing, namely faster threads do not remain idle while waiting for slower threads to complete execution.

1.3 Measuring up peak performance

The peak performance achieved by our algorithm is 8.06 GigaFlops/s, which is about 9.5 slower than the peak performance of one hive machine. This is because our algorithm only exploits multiple threads of execution, but does not exploit memory optimisations and SSE intrinsics. I would do some loop unrolling, reorder the loops so that I can load some values out of the inner loop and save them on some register, and use SSE intrinsics to do multiple instructions at the same time.

1.4 Preference

I like OpenMP better because it is much more productive that pThreads. Id est development is faster because a lot is done for you already, and there is less risk to shoot yourself in the foot compared to pThreads. However, sometimes I prefer to have more control on how my threads are being utilised and for that I would use pThreads.

1.5 How does OpenMp work

OpenMP is a high level abstraction that uses pThreads under the hood. When the -fopenmp flag is activated the compiler transforms the source code into multithreaded code (like in the example below), and follows roughly the following steps:

1. Reads and parses OpenMP directives, and checks for correctness
2. Translates SECTIONs to DO/FOR statements
3. Makes implicit barriers explicit
4. Implements some optimisations like merging adjacent parallel regions
5. Implements variable data attributes, sets up storage and range for initialisation

Listing 1: Original Code

```
#pragma omp for
for( i = 0; i < n; i++ )
{  }
```

Listing 2: Transformation

```
tid = ompc_get_thread_num();
ompc_static_init (tid, lower,upper,
incr,.);
for( i = lower;i < upper;i += incr )
{   }
// Implicit BARRIER
ompc_barrier();
```

2 Two-Dimensional Convolution

2.1 Scaling Plots

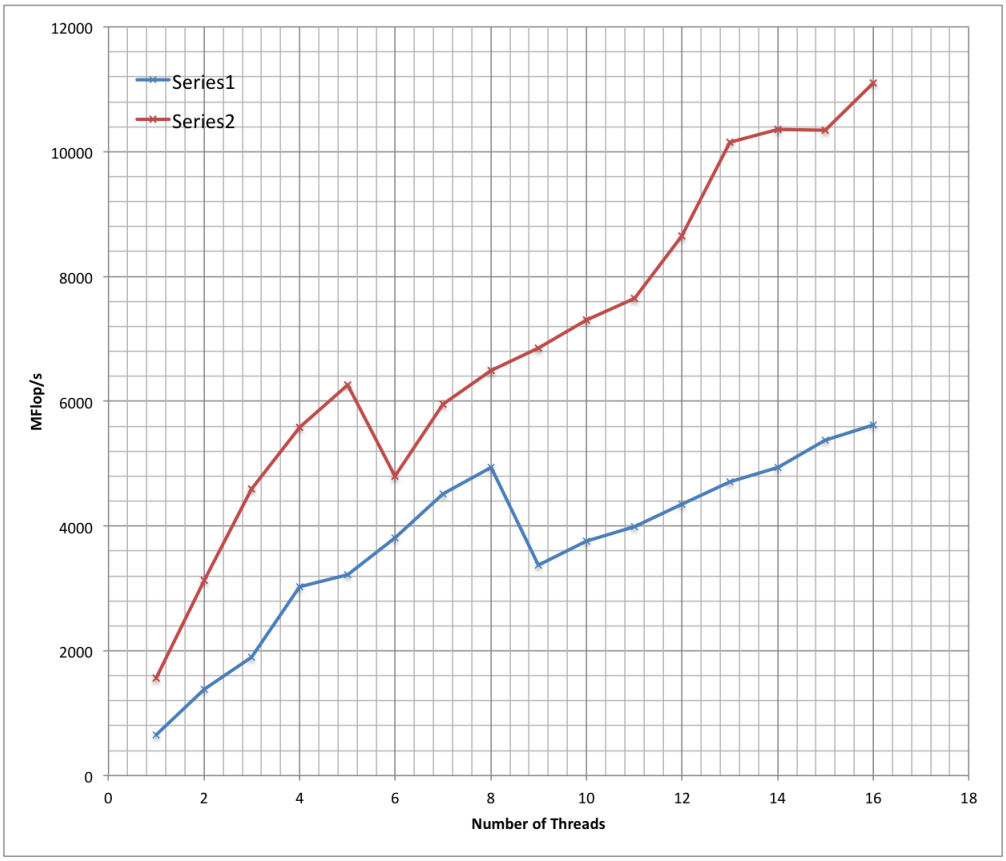

Figure 2: series 1: -n1; series 2: -n10

2.2 Comments On Scaling Performances

The code scales better with maximum radius 10. This could be due to the fact that we exploit more spatial locality in the radius loops. We could improve performance by coin multiple operations with SSE intrinsics. For example when the blur radius is even.

2.3 Time spent on assignment

About 4 hours...