OpenMP Lab Assignment

Jan-Hendrik Niemann and Andreas Radke

November 28, 2017

1 Code Changes

To parallelize the Laplace-Diffusion code we had to implement three little changes which we want to present in the following section.

1. The first change consists of moving the *iter* and *error* variable to the top of the code and make them *threadprivate*.

   "*The threadprivate directive is used to make global file scope variables (C/C++) local and persistent to a thread through the execution of multiple parallel regions*", (compare [1]).

   We use this to control *iter* and *error* during the *while*-loop.

```
#include <stdlib.h>

int iter;
float error;
#pragma omp threadprivate(iter,error)
```

2. The second change is a simple *omp for* pragma to parallelize the *for*-loop. Note that we do not create (or terminate) the parallel environment for this *for*-loop. This takes place in change 3.

```
float laplace_step(float *in, float *out, int n)
{
  int i, j;
  float error=0.0f;
  #pragma omp for
  for ( j=1; j < n-1; j++ )
 #pragma omp simd reduction(max:error)
  for ( i=1; i < n-1; i++ )
    {
      out[j*n+i]= stencil(in[j*n+i+1], in[j*n+i-1],
                          in[(j-1)*n+i], in[(j+1)*n+i]);
      error = max_error( error, out[j*n+i], in[j*n+i] );
    }
  return error;
}
```

3. In here we are creating the parallel environment while we are keeping the pointers *A* and *temp firstprivate*. This pragma creates private variables for each thread with an automatic initialization (with the value from the original variable).

   copyin lets us assign the same value to *threadprivate* variables (see above) for all threads (see [1]).

```
#pragma omp parallel firstprivate(A, temp) copyin(iter, error)
while ( error > tol*tol && iter < iter_max ){
  iter++;
  error= laplace_step (A, temp, n);
  float *swap= A; A=temp; temp= swap; // swap pointers A & temp
}
```

2 Perf Analysis

In Figure 1 we can see that the utilized CPUs strictly (and in strong correspondence) increases with the number of used threads.

But unfortunately it does not display an improvement in wall clock time outputted by perf. The data points seem close to each other and a bit erratic with no clear trend (compare Figure 2). We executed it several times (maybe there was some load on the server by other users but the utilized CPU usage does not indicate it) but there was never a clear trend to observe.

Furthermore decreasing instructions per cycle (or increasing instructions, because the cycles are roughly constant) seems to offset the advantage in used CPUs. Additionally to the increasing cycles the cache-misses grow rapidly too.

3 TAU Tracing

In Figure 4 we can see that the execution time declines with more threads. We took the mean run time per thread for the *for*-loop which takes up the most part of the computation. For 1, 2, and 4 threads the execution times seems almost perfectly halved. Only the step to 8 threads does not give such an improvement. This may be due to additional load on the server or due to less efficient Hyperthreading scheduling (the Intel processor only has four real cores and emulates four additional ones via Hyperthreading).

4 PAPI

We analyzed our program with the PAPI counters *PAPI_L2_TCM*, *PAPI_L3_TCM* and *PAPI_FP_OPS* which corresponds to L2 and L3 cache misses respectively and the num-

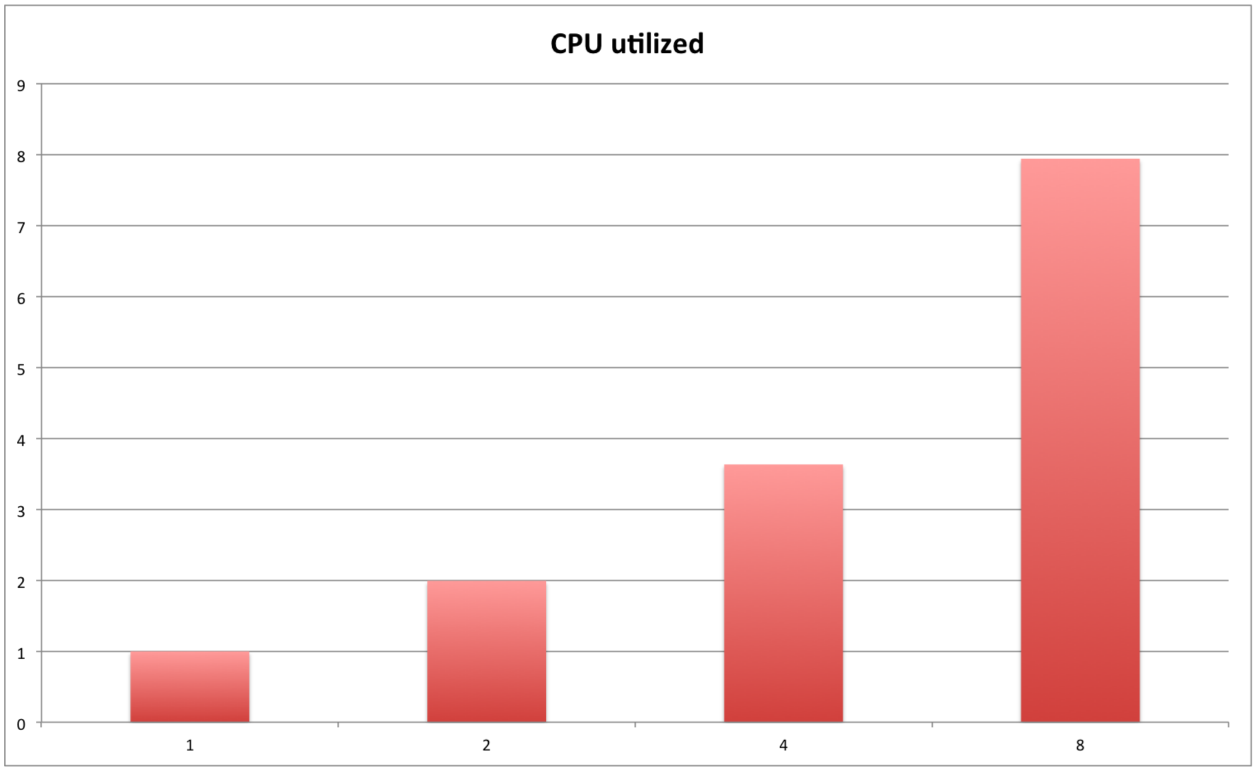

Figure 1: CPU utilized

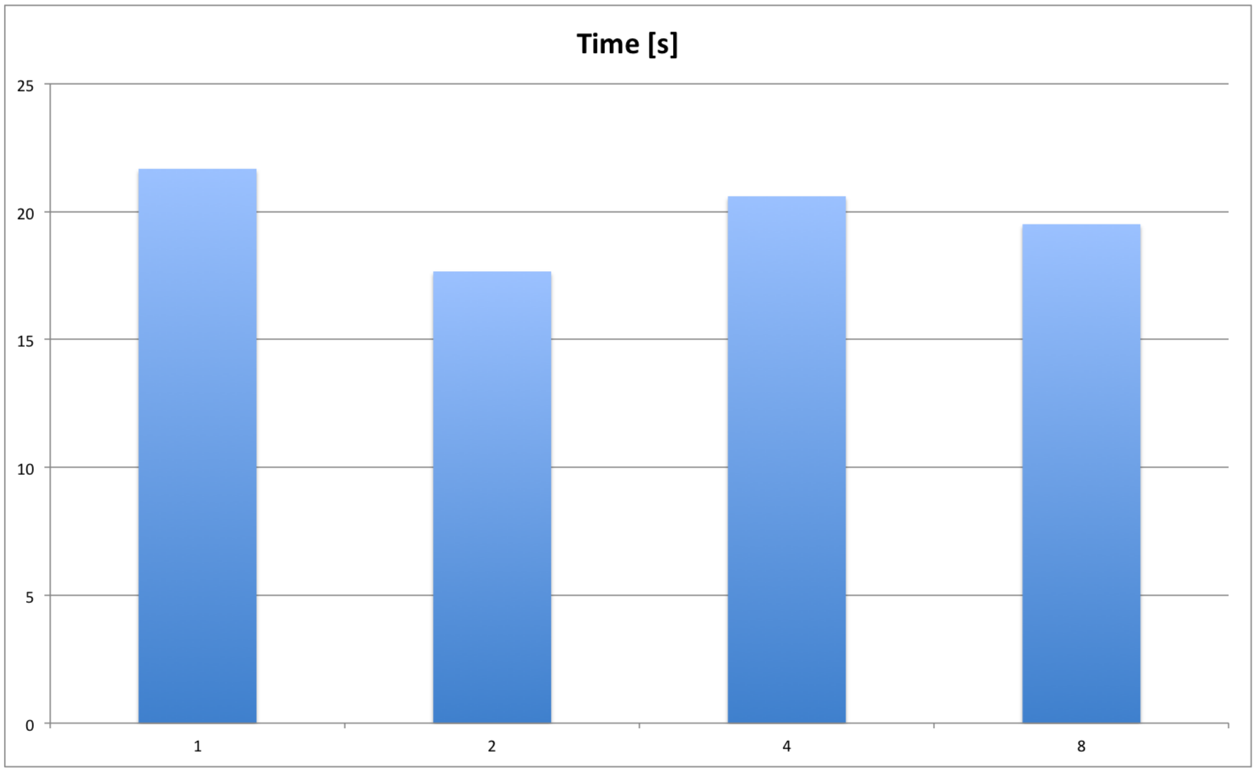

Figure 2: Wall clock time output in seconds by perf

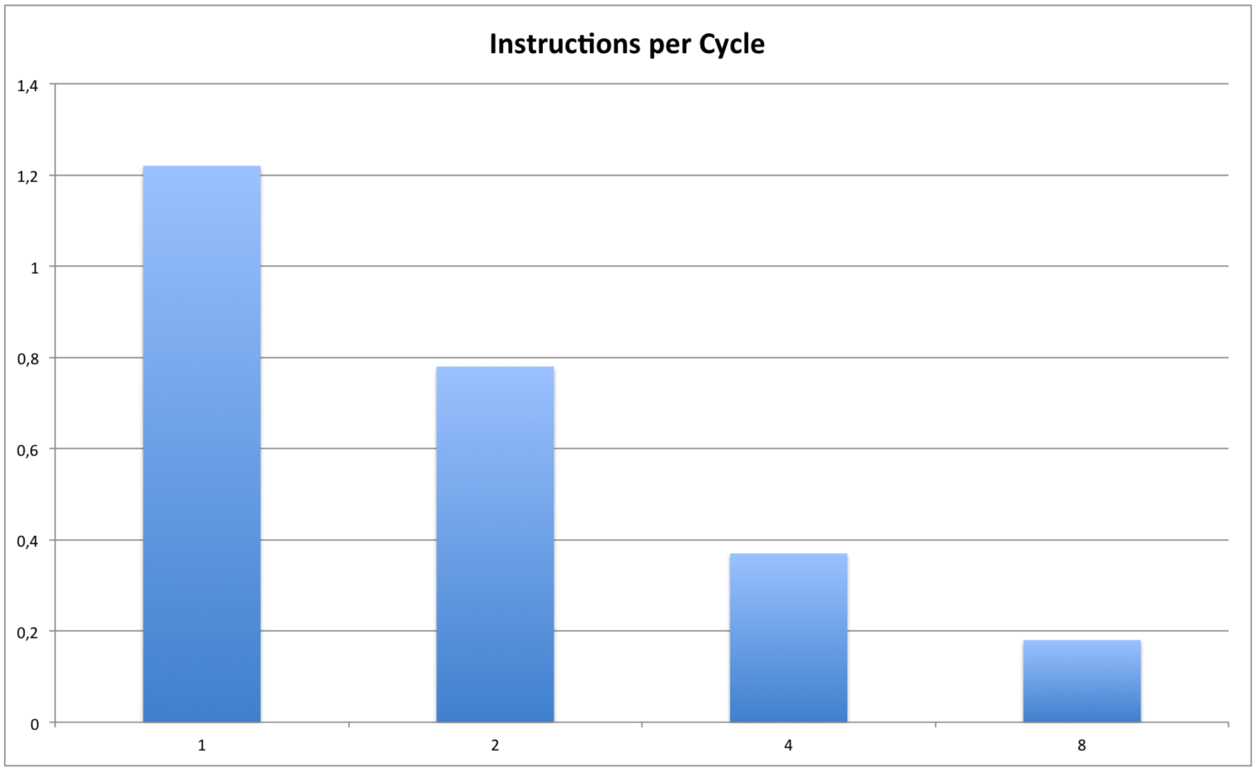

Figure 3: Instructions per Cycle

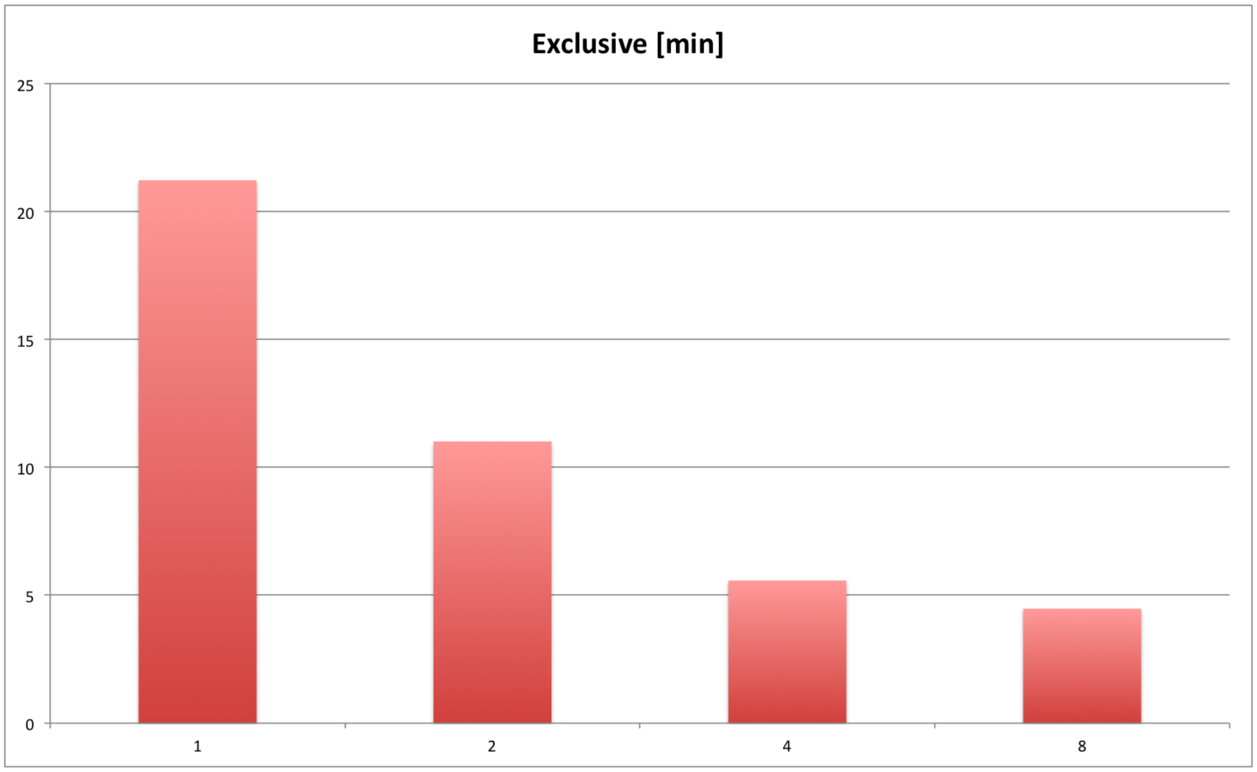

Figure 4: Tau Execution Time in Minutes: for (loop body)

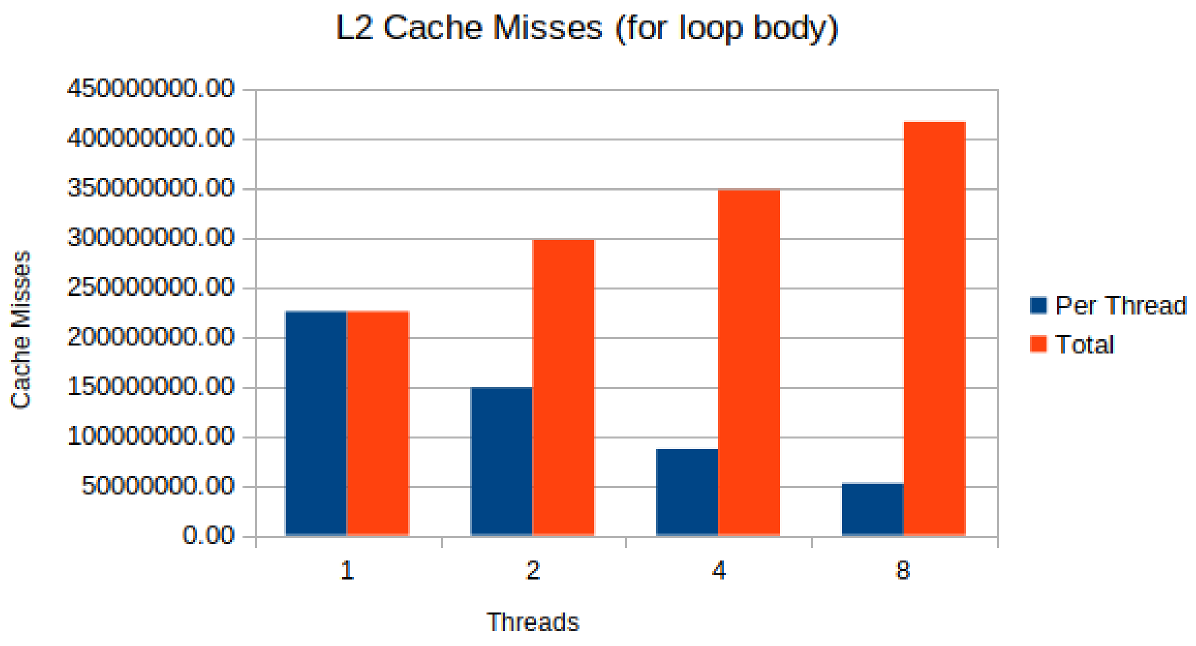

Figure 5: Number of L2 Cache Misses in the for-loop

ber of floating point operations.

The machine on which our program was running has a quad-core *Core i7-950* CPU with four times 256 KiByte L2 cache and 8.192 KiByte shared L3 cache. As you can see in figure 6 the number of L2/L3 cache misses per thread decreases with increasing number of threads. For the L2 case it is obvious that the number decreases because each core has its own L2 cache (but it is not that obvious for the case from 4 to 8 threads). Due to some overhead for each thread the number of misses is not perfectly halved and the total number of all L2 misses grows.

For L3 there is not a clear growth of total L3 misses. This might be due to the fact that a single threaded program has less (total) cache than a multi-threaded program in sum. Each core has its own L1 and L2 cache which are used first but the L3 cache is shared. This not clear behavior can be due to uncertainty if a thread is running on a different core or if two threads are mostly run by one core (which is definitely true for 8 threads).

In Figure 7 one can see that the threaded program behaves almost *perfectly* in regard to floating point operations. That is if the number of threads doubles one thread has to do only half the number of floating point operations. Although the total number of FP ops stays the same (with a slight increase for more threads). The tau results let us conclude that the computational reduction for the parallelized program works indeed quite well.

Resources

[1] `https://computing.llnl.gov/tutorials/openMP/`

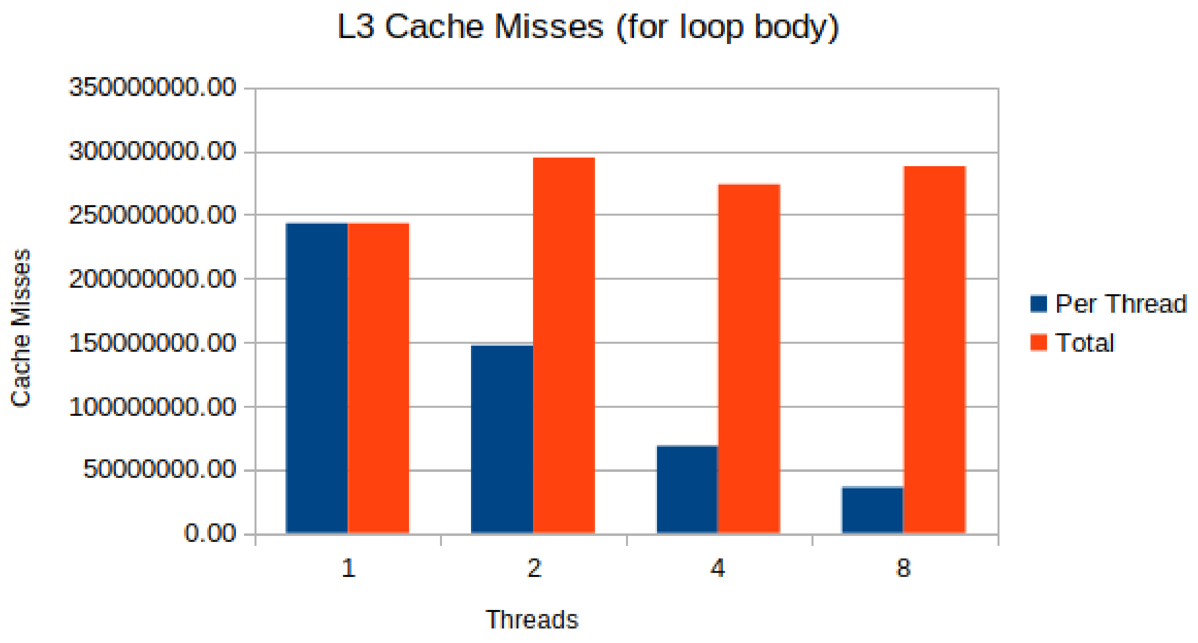

Figure 6: Number of L3 Cache Misses in the for-loop

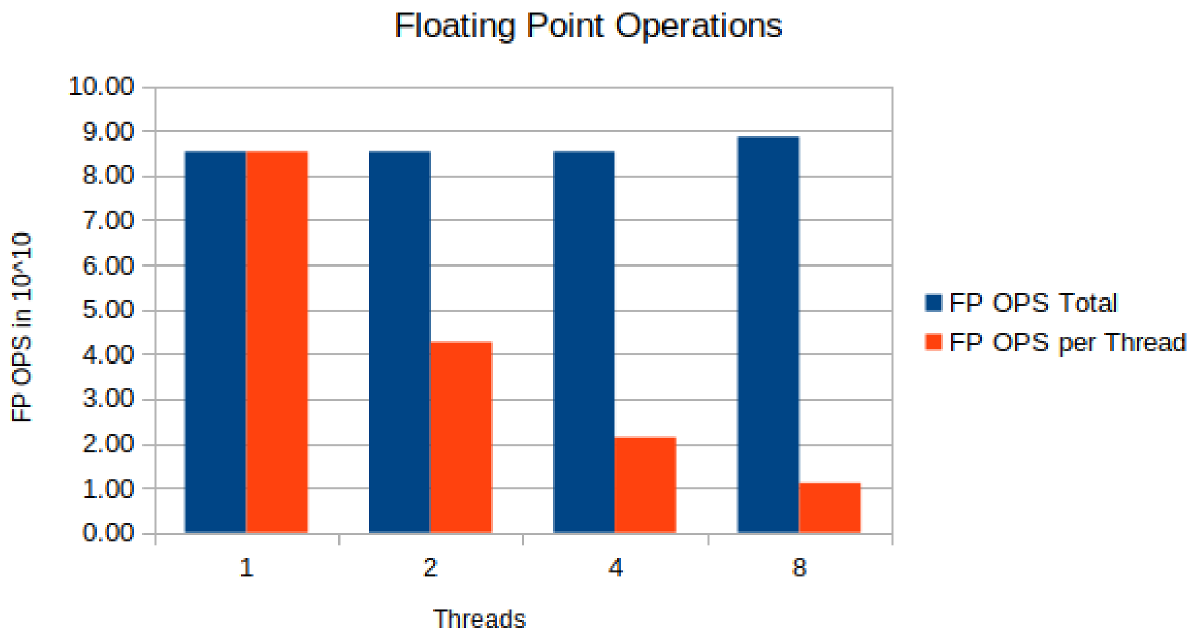

Figure 7: Floating Point Operations in 10^{10}