Lab Course: Efficient Programming of Multicore Processors and Supercomputers

Jonas Mayer, Paul Preißner, Konrad Pröll

May 30, 2017

3.2 Automatic Parallelization

3.2.1 Try the automatic parallelization feature of icc

ICC's automatic parallelization feature is switched on by the compiler flag -parallel. Thus, we added this flag to the flags in the Makefile.

3.2.2 The compiler might parallelize all the loops. With the "-opt-report-phase=par" option you will receive a report explaining the reasons why parallelization was not possible.

Initially, the compiler suggests there is a parallel dependence between iterations of the outer loop and thus does not parallelize it. Obviously, unew, diff and sum are being reused between different iterations, so we might need to fix this by inserting the corresponding pragma.

3.2.3 Try to improve the parallelization by rewriting the code and inserting pragmas (no OpenMP pragmas). Explain why it worked or did not work.

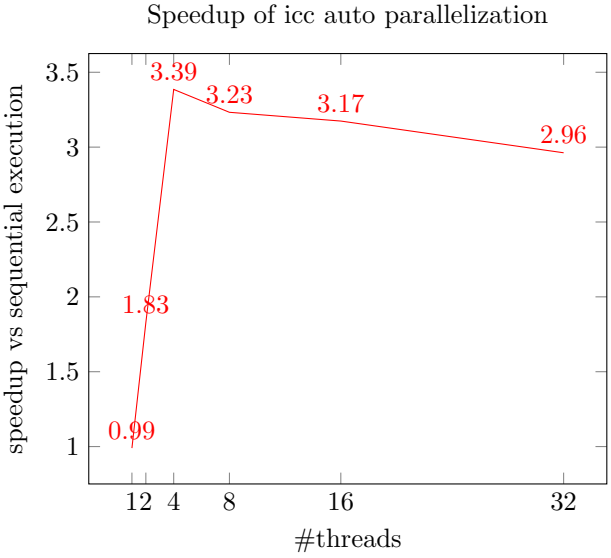
By putting "#pragma parallel private (unew, diff, sum)" ahead of the outer loop, we suggest to the compiler that this loop might get parallelized and avoid the issue of variables being written into accross different threads. However, because the loopsize cannot be determined at compiletime, icc will not parallelize the llop due to "insufficient computational work".
If we change the pragma to "#pragma parallel always private (unew, diff, sum)", the loop will be parallelized. You can also force the compiler to parallelize by adding "-par-threshold0" to the compiler flags. This obviously does not affect specific loops, but changes the global setting in your program.
The flag -guide-par gives you hints on sections of the code that might get parallelized.

```
double relax_jacobi( double **u1, double **utmp1,
          unsigned sizex, unsigned sizey )
{ [...]
#pragma parallel always
  for( i=1; i<sizey-1; i++ ) {
        int ii=i*sizex;
        int iim1=(i-1)*sizex;
        int iip1=(i+1)*sizex;

#pragma ivdep
    for( j=1; j<sizex-1; j++ ){
       unew = 0.25 * (u[ ii+(j-1) ]+
                                    u[ ii+(j+1) ]+
                                    u[ iim1+j ]+
                                    u[ iip1+j ]);
                    diff = unew - u[ii + j];
                    utmp[ii+j] = unew;
                    sum += diff * diff;

    }
  } [...]
}
```

3.2.4 Do performance measurements for 1, 2, 4, 8, 16 and 32 threads on SuperMUC with the configuration in test.dat. Provide a speedup graph for problem size 5200



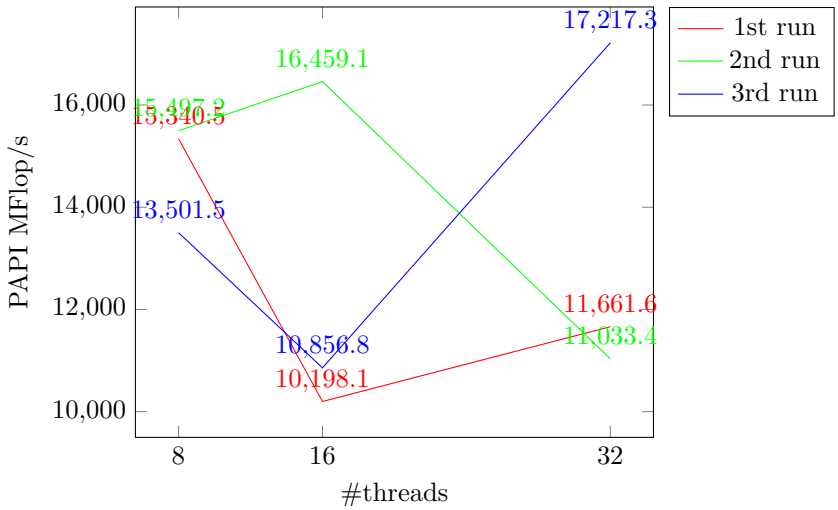
3.2.5 Is there a difference between the sequential time from Assignment 3.1 and the sequential time of the OpenMP version?

As one can see in the plot, the parallelized version for one thread is slightly slower. This is probably due to overhead.

Addendum

We noticed that the runtime, even in batchscripts, is pretty inconsistent.

Mflop/s for 8/16/32t for different runs at 5200 res



Also, more threads do not necessarily speed up, as we, in particular in this application, which mainly uses floating point operations, do not gain anything when we use Hyperthreading. `https://www.researchgate.net/publication/267242498_An_Empirical_Study_of_Hyper-Threading_in_High_Performance_Computing_Clusters`

EDIT: More thoughts on for the poor speedup and possible reasons like memory bandwidth can be found in section 3.3.4.