# Lab Course: Efficient Programming of Multicore Processors and Supercomputers

Jonas Mayer, Paul Preißner, Konrad Pröll

May 30, 2017

## 3.3 OpenMP

### 3.3.1 Parallelize the Jacobi algorithm with OpenMP

```
#pragma omp parallel
  {
#pragma omp for schedule (guided) private (unew,
   diff) reduction (+:sum) nowait
    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;
      }
    }
  }
```

#Pragma omp parallel creates a parallel region. #Pragma omp for shares the work among the threads. Each thread writes into diff and unew and reads from them, so we have to declare them as private data. Every thread adds some value into sum (which is our residuum), so we have to use reduction for it. We specify nowait because #pragma omp parallel already introduces a barrier and there is no additional value in having a barrier after #pragma omp for.

### 3.3.2 Optimize for NUMA according to first touch allocation policy

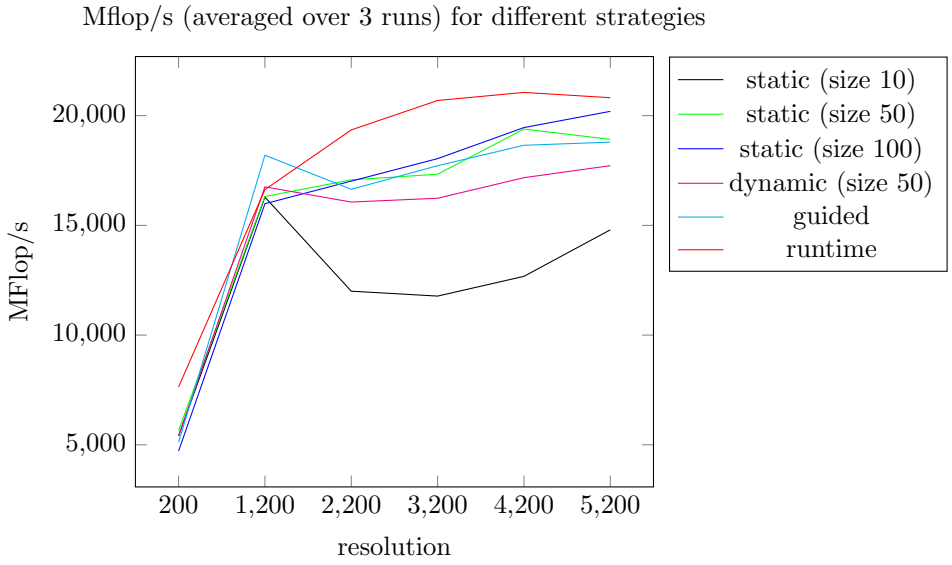
First touch allocation policy means that the operating system allocates memory upon initialization in the memory that is nearest to the allocating cpu. Thus, in a NUMA system, if we just use one cpu to initialize the arrays, the other cpu will have a higher latency when accessing its data. We can avoid this by using the very same OpenMP directives for allocating the memory as we used to calculate the new values.

```
int initialize( algoparam_t *param )
{
//...
#pragma omp parallel
    {
#pragma omp for schedule (guided) nowait
      for (i=0;i<np;i++){
        for (j=0;j<np;j++){
          param->u[i*np+j]=0;
          param->uhelp[i*np+j]=0;
        }
      }
    }
//...
}
```

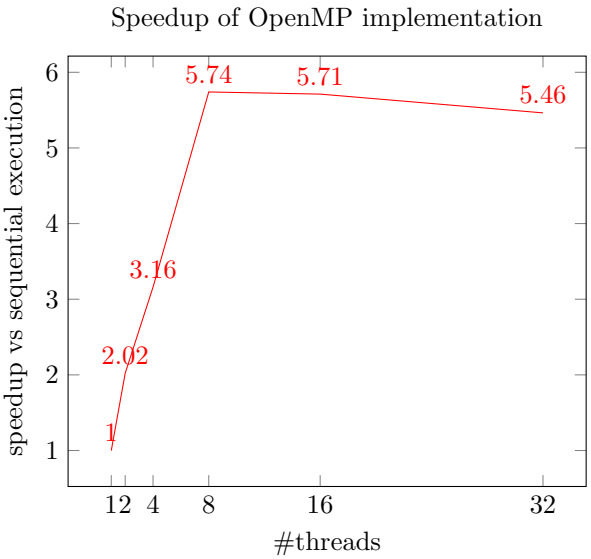
Source: http://www.nersc.gov/users/computational-systems/cori/application-porting-and-perf
improving-openmp-scaling/#toc-anchor-3

### 3.3.3 Any other optimization possibilities?

We checked the performance change for the various scheduling strategies. We tested static at sizes 10, 50 and 100, dynamic at size 50, guided and runtime. Looking at the numbers achieved, runtime seems to be the fastest strategy, followed by guided, static size 50 and static size 100 on roughly the same level, with dynamic size 50 in second-to-last and static size 10 in last spot by a wide margin. Thus, we decided to use scheduling strategy runtime for the measurements in the next task.




### 3.3.4 Run problem size 5200 with 1, 2, 4, 8, 16, 32 threads. Give the speedup diagram and explain your findings.




As opposed to the icc auto parallelisation the speedup peaks at 8 threads with a speedup at roughly 5.7. Prior to NUMA optimizations no speedup improvement was observable beyond 4 threads with OpenMP. Our observations have shown that the different implementations have a peak performance of 12 MFLOP/S (icc, OMP no NUMA) and 24 MFLOP/S (OMP NUMA), which would be, according to coarse estimates, roughly enough to saturate the mem-

ory bandwidth of 1 and 2 processors respectively (102.4 GB/s per processor). This shows that the NUMA optimization met its goal to make full use of both sockets in a node. A rough estimation would be, that the memory bandwidth divided by the floprate on one cpu (i. e. prior to the optimization according to first touch allocation policy) is approximately 8, which actually is sizeof(double). That means, that in order to reach the memory limit, we would need to just have one memory access for a double per floating point operation, which sounds pretty realistic. Due to this and due to the overhead, the speedup gets worse with more threads, because the application is limited by the memory bandwidth.