Lab Course Efficient Programming of Multicore Processors and Supercomputers

1.1.

1. Show the initial performance data.

Jacobi:

Resolution	MFlop/s
100	475.407468
300	410.097573
500	389.720385
700	345.743325
900	266.359908
1100	245.029018
1300	241.159283
1500	228.631083
1700	217.060715
1900	208.342824
2100	216.940421
2300	189.397070
2500	180.502628
2700	185.770487
2900	176.501386

It seems to stabilize at around 180 MFlop/s.

Gauss-Seidel: 591.716844 MFlop/s

2. Explain how the FLOP/s metric is measured. Which floating point operations are taken into account?

In heat.c, the number of iterations is counted and multiplied with the resolution of the grid and the factor 11, meaning that there are 11 Floating Point Operations per grid point and iteration step. To see which Floating Point Operation are taken into account, we have to separate between the two algorithms:

Jacobi:

Gauss-Seidel: In void relax_gauss/relax_jacobi, there are 4 FLops per grid point: Three additions to get the sum of the four points surrounding the calculated grid point, and a multiplication with 0.25, to get the mean. In double residual_gauss/residual/jacobi, there

is 7 Flops: The same four in the calculation of unew, plus one in the calculation of the value to be assigned to diff, (line 41), the value of diff*diff and the sum of that value and sum (both line 42)

1.2

What is the meaning of -ipo and -fno-alias?

ipo is an acronym that means Interprocedural Optimization. When used, ipo analyzes code for various optimizations. A full list canbe found at https://software.intel.com/en-us/node/522667

fno-alias

Forces the compiler to assume no aliasing

What is the meaning of "ivdep"?

Ivdep is an abbreviation for ignore vector dependencies. It is a pragma to be inserted into the code before loops a compiler might not vectorize because it assumes that there is dependency. On the intel documentation, the example

```
void ignore_vec_dep(int *a, int k, int c, int m) {
  #pragma ivdep
  for (int i = 0; i < m; i++)
    a[i] = a[i + k] * c;
}
```
(Source: https://software.intel.com/en-us/node/524501)

Is used, where a compiler under normal circumstances would not vectorizer the loop, because the sign of k is not known. This only affects assumed dependencies, but not proven dependencies.

The Intel compiler provides reports when using "opt-report" option. What does it print out, and what does it mean?

The compiler prints out optimization reports, which show which part of the program could be optimized. It shows for each loop whether it was vectorized or not and, if not, explains why. For example, for the function void relax_jacobi, the output is:

```
================================================================================
Begin optimization report for: relax_jacobi(double *, double *, unsigned int, unsigned int)
    Report from: Interprocedural optimizations [ipo]
INLINE REPORT: (relax_jacobi(double *, double *, unsigned int, unsigned int)) [2] relax_jacobi.c(43,1)
    Report from: Loop nest, Vector & Auto-parallelization optimizations [loop, vec, par]
LOOP BEGIN at relax_jacobi.c(46,5)
```

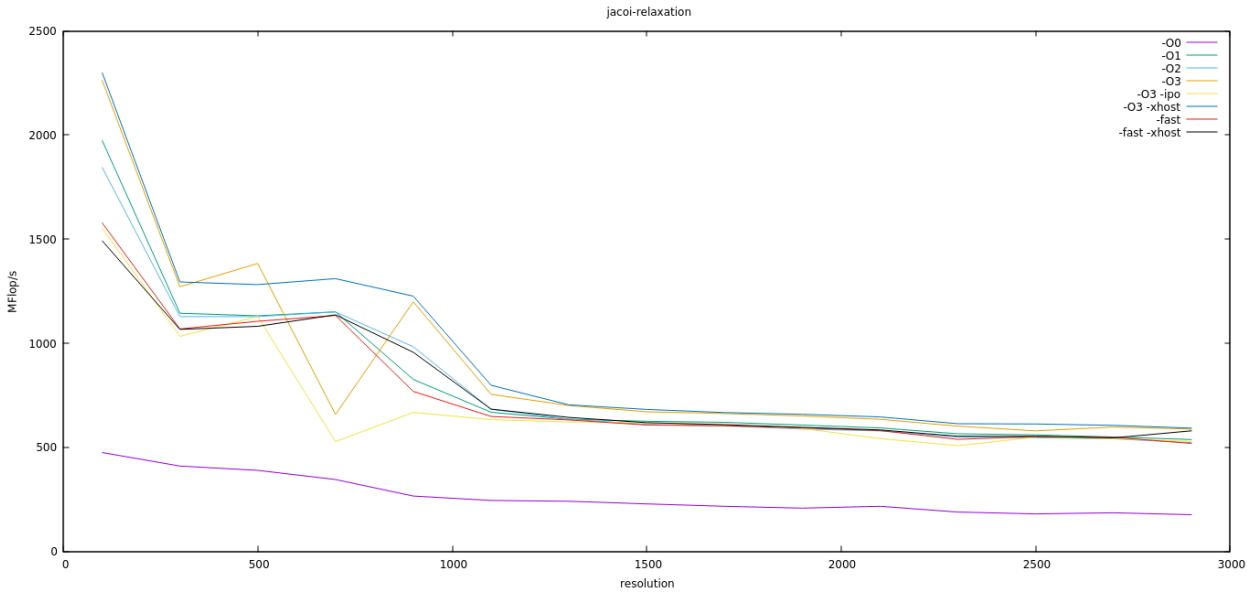
```
   remark #15344: loop was not vectorized: vector dependence prevents vectorization. First dependence is shown
below. Use level 5 report for details

   remark #15346: vector dependence: assumed OUTPUT dependence between utmp line 50 and utmp line 50

   LOOP BEGIN at relax_jacobi.c(48,2)

      remark #15344: loop was not vectorized: vector dependence prevents vectorization. First dependence is shown
below. Use level 5 report for details

      remark #15346: vector dependence: assumed FLOW dependence between utmp line 50 and u line 50

   LOOP END

LOOP END

LOOP BEGIN at relax_jacobi.c(59,5)

   remark #15344: loop was not vectorized: vector dependence prevents vectorization. First dependence is shown
below. Use level 5 report for details

   remark #15346: vector dependence: assumed OUTPUT dependence between u line 63 and u line 63

   LOOP BEGIN at relax_jacobi.c(61,2)

      remark #15344: loop was not vectorized: vector dependence prevents vectorization. First dependence is shown
below. Use level 5 report for details

      remark #15346: vector dependence: assumed FLOW dependence between u line 63 and utmp line 63

      remark #25439: unrolled with remainder by 2

   LOOP END

   LOOP BEGIN at relax_jacobi.c(61,2)

   <Remainder>

   LOOP END

LOOP END
```

One can see, that, the first loop was not vectorized due to an assumed output (WAW) Dependence in line 50 and the second loop was not vectorized due to an assumed flow (RAW) dependence in line 63, but was unrolled by 2. Those hints can be used to set the ivdep pragmas.

Is the code vectorized by the compiler?

Partially. The inner loop of residual_jacobi is vectorized, while no part of the gauss-relaxation is vectorized.

What is the performance result of these options. Present a graph!



1.3

Submit you batch script and the job output from 1.3

Submitted in separate files.

Does the performance differ to a run on the login node?

Generally, it does not. However, sometimes the performance on the login node is a lot slower than usual, perhaps because there is someone else running some program.

1.4

Compile with "-p -g" for use with gprof. A run will produce the file "gmon.out" now. View profiling results with "gprof ./heat gmon.out".

Copy the output of gprof into your answer.

Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
81.55	101.50	101.50	750	135.33	135.33	relax_jacobi
18.12	124.06	22.56	750	30.08	30.08	residual_jacobi
0.22	124.33	0.27	1	270.00	270.00	write_image
0.05	124.39	0.06	15	4.00	4.00	initialize
0.04	124.44	0.05	1	50.00	50.00	coarsen
0.02	124.47	0.03				__libm_pow_e7

```
  0.00    124.47     0.00       30     0.00     0.00  wtime
  0.00    124.47     0.00       15     0.00     0.00  finalize
  0.00    124.47     0.00        1     0.00     0.00  print_params
  0.00    124.47     0.00        1     0.00     0.00  read_input


//explanation for the output
Call graph (explanation follows)
granularity: each sample hit covers 4 byte(s) for 0.01% of 124.47 seconds
index % time    self  children    called     name
                                             <spontaneous>
[1]    100.0    0.00  124.44                 main [1]
               101.50    0.00     750/750         relax_jacobi [2]
                22.56    0.00     750/750         residual_jacobi [3]
                 0.27    0.00       1/1           write_image [4]
                 0.06    0.00      15/15          initialize [5]
                 0.05    0.00       1/1           coarsen [6]
                 0.00    0.00      30/30          wtime [8]
                 0.00    0.00      15/15          finalize [9]
                 0.00    0.00       1/1           read_input [11]
                 0.00    0.00       1/1           print_params [10]
-----------------------------------------------
               101.50    0.00     750/750         main [1]
[2]     81.5  101.50    0.00     750         relax_jacobi [2]
-----------------------------------------------
                22.56    0.00     750/750         main [1]
[3]     18.1   22.56    0.00     750         residual_jacobi [3]
-----------------------------------------------
                 0.27    0.00       1/1           main [1]
[4]      0.2    0.27    0.00       1         write_image [4]
-----------------------------------------------
                 0.06    0.00      15/15          main [1]
[5]      0.0    0.06    0.00      15         initialize [5]
```

```
-----------------------------------------------
                0.05    0.00       1/1          main [1]
[6]      0.0    0.05    0.00       1         coarsen [6]
-----------------------------------------------
                                    <spontaneous>
[7]      0.0    0.03    0.00                __libm_pow_e7 [7]
-----------------------------------------------
                0.00    0.00      30/30          main [1]
[8]      0.0    0.00    0.00      30         wtime [8]
-----------------------------------------------
                0.00    0.00      15/15          main [1]
[9]      0.0    0.00    0.00      15         finalize [9]
-----------------------------------------------
                0.00    0.00       1/1           main [1]
[10]     0.0    0.00    0.00       1         print_params [10]
-----------------------------------------------
                0.00    0.00       1/1           main [1]
[11]     0.0    0.00    0.00       1         read_input [11]
-----------------------------------------------
```

//explanation

Index by function name

```
   [7] __libm_pow_e7          [10] print_params           [4] write_image
   [6] coarsen                [11] read_input             [8] wtime
   [9] finalize               [2] relax_jacobi
   [5] initialize             [3] residual_jacobi
```

What is the run-time overhead of "-p"?

Time measurement spent in each function, tree of function calls, counter of function executions. The overhead can sometimes be more than 260% of the actual execution.

Source: http://gernotklingler.com/blog/gprof-valgrind-gperftools-evaluation-tools-application-level-cpu-profiling-linux/

Which functions take most of time?

The actual relaxations, as one would expect due to the higher number of Flops, followed by the calculations of the residua.