Lab Course
Efficient Programming of Multicore Processors and Supercomputers

Report 1: Compiler Optimizations

Jonas Mayer, Paul Preißner, Konrad Pröll

Fakultät für Informatik
Technische Universität München

May 11th 2017

Task 1.1 Understanding the code

- 1. Show the initial performance data.
- 2. Explain how the FLOP/s metric is measured. Which floating point operations are taken into account?

1.1.1. Show the initial performance data

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

1.1.1. Show the initial performance data

- It seems to stabilize at around 180 MFlop/s
- Gauss-Seidel achieves 591.7 MFlop/s

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

```
1         // Flop count after <i> iterations
2         flop = iter * 11.0 * param.act_res * param.act_res;
```

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

```
1  int i, j;
2  for( j=1; j<sizex-1; j++ )
3  {
4  for( i=1; i<sizey-1; i++ )
5      {
6      utmp[i*sizex+j]= 0.25 * (u[ i*sizex      + (j-1) ]+
7                               u[ i*sizex      + (j+1) ]+
8                                       u[ (i-1)*sizex + j
                                           ]+
9                               u[ (i+1)*sizex + j      ]);
10 }
11 }
```

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

```
1  double unew, diff, sum=0.0;
2  for( i=1; i<sizey-1; i++ )
3  {
4  for( j=1; j<sizex-1; j++ )
5            {
6       unew = 0.25 * (u[ i*sizex     + (j-1) ]+
7                      u[ i*sizex     + (j+1) ]+
8                      u[ (i-1)*sizex + j     ]+
9                      u[ (i+1)*sizex + j     ]);
10           diff = unew - u[i*sizex + j];
11           sum += diff * diff;
12 }
13 }
```

Task 1.2 Compiler Options

- What is the meaning of -ipo and -fno-alias
- What is the meaning of “ivdep”?
- The Intel compiler provides reports when using “opt-report” option. What does it print out, and what does it mean?
- Is the code vectorized by the compiler?
- What is the performance result of these options. Present a graph!

Task 1.2 Compiler Options

- -ipo:
  - “Interprocedural Optimization”
  - analyzes the code and applies various (e.g. inlining)
  - full list can be found at `https://software.intel.com/en-us/node/522667`
- -fno-alias
  - forces compiler to assume no aliasing
  - aliasing: accessing one memory cell through different symbolic names

Task 1.2 Compiler Options

- ivdep:
  - “ignore vector dependencies”
  - pragma to be inserted into code
  - compiler might assume a loop non vectorizable
  - #pragma ivdep tells compiler that there is no dependency
  -

```
1 void ignore_vec_dep(int *a, int k, int c, int m) {
2         #pragma ivdep
3         for (int i = 0; i < m; i++)
4                 a[i] = a[i + k] * c;
5 }
```

Task 1.2 Compiler Options

- opt-report
  - optimization reports by the compiler
  - show which part of the code was optimized
  - for not optimized parts, also shows the reason why it could not be optimized

Task 1.2 Compiler Options

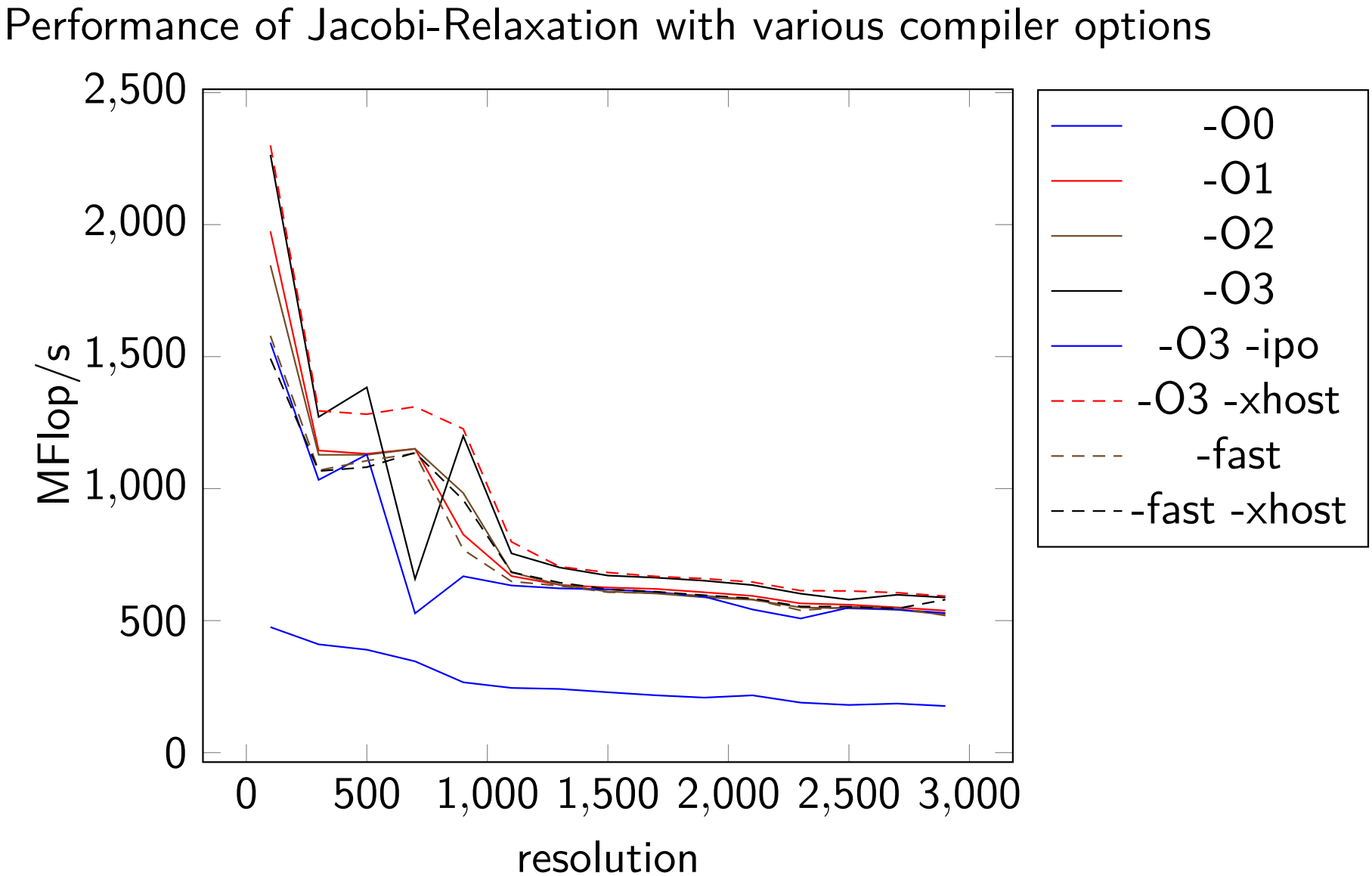
```
1 Begin optimization report for: relax_jacobi(double *,
      double *, unsigned int, unsigned int)
2         Report from: Interprocedural optimizations [ipo]
3 INLINE REPORT: (relax_jacobi(double *, double *, unsigned
      int, unsigned int)) [2] relax_jacobi.c(43,1)
4         Report from: Loop nest, Vector &
              Auto-parallelization optimizations [loop, vec,
              par]
5 LOOP BEGIN at relax_jacobi.c(46,5)
6         remark #15344: loop was not vectorized: vector
              dependence prevents vectorization. First
              dependence is shown below. Use level 5 report
              for details
7         remark #15346: vector dependence: assumed OUTPUT
              dependence between utmp line 50 and utmp line 50
8 LOOP END
```

Task 1.2 Compiler Options

- Is the code vectorized by the compiler?
  - Only the inner part of residual_jacobi is vectorized
  -

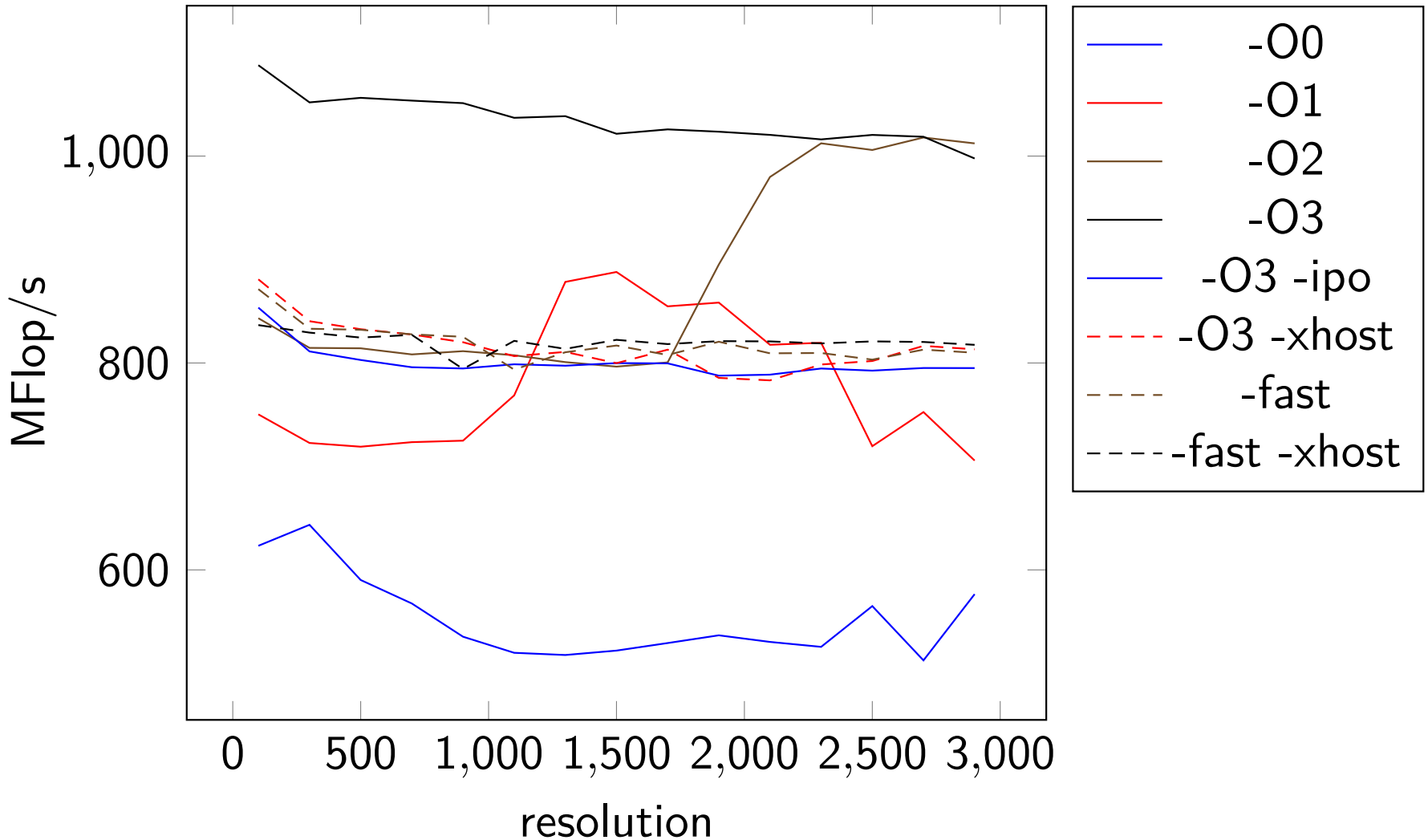
```
1  Begin optimization report for:
       residual_jacobi(double *, unsigned int,
       unsigned int)
2      Report from: Interprocedural optimizations
           [ipo]
3  LOOP BEGIN at relax_jacobi.c(20,5)
4     remark #15542: loop was not vectorized: inner
          loop was already vectorized
5
6     LOOP BEGIN at relax_jacobi.c(22,2)
7        remark #15300: LOOP WAS VECTORIZED
8     LOOP END
9
10    LOOP BEGIN at relax_jacobi.c(22,2)
11    <Remainder loop for vectorization>
12    LOOP END
13 LOOP END
```

Task 1.2 Compiler Options



Task 1.2 Compiler Options

Performance of GS-Relaxation with various compiler options



Task 1.3: Batch Script

```
#!/bin/bash
#@ wall_clock_limit =00:2:00
#@ job_name = praktikum_testjob
#@ job_type = parallel
#@ class = test
#@ energy_policy_tag= NONE
#@ output = job$(jobid).out
#@ error = job$(jobid).out
#@ node = 1
#@ network.MPI = us
#@ island_count = 1
#@ total_tasks = 1
#@ queue
. /etc/profile
cd /gpfs/work/h039v/h039vaj
poe ./heat test.dat
```

Task 1.3: Results.out

```
1  DISK GROUP QUOTAS for home and project file systems:
2  Filesystem                                        Quota      Used
       Space      Free Space
3  /home/hpc/h039v              ($HOME)          102.4GB   355.9MB (
       0%)   102.1GB (100%)
4  /gpfs/work/h039v             ($WORK)         1000.0GB    96.2MB
       ( 0%)   999.9GB (100%)
5  ------------------------------------------------------------------------
6
7  Executing LRZ User Prolog ...
8  Resolutions        : (100, 300, ... 2900)
9  Iterations         : 50
10 Algorithm          : 0 (Jacobi)
11 Num. Heat sources : 2
12    1: (0.00, 0.00) 1.00 1.00
13    2: (1.00, 1.00) 1.00 0.50
14
15 ...
```

Task 1.3

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

Generally it does not. However, sometimes the performance on the login node is unusually slow, most likely because someone else is running a program.

Task 1.4 Gprof

- What is the run-time overhead of "-p"?
  - Time measurement spent in each function, tree of function calls, counter of function executions. The overhead is estimated to be up to 260% of the actual execution.

    Source: `http://gernotklingler.com/blog/gprof-valgrind-gperftools-evaluation-tools-application-`
- Which functions take most of the time?
  - The actual relaxations *(e.g. 135.33 ms/call of relax_jacobi)* as one would expect due to the higher number of Flops, followed by the calculations of the residuals *(e.g. 30.08 ms/call of residual_jacobi)*.

Task 1.4 Gprof - gmon.out of run after "-p -g" compile 1/4

```
Flat profile:
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  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
```

Task 1.4 Gprof - gmon.out of run after "-p -g" compile 2/4

```
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]
-----------------------------------------------
```

Task 1.4 Gprof - gmon.out of run after "-p -g" compile 3/4

```
                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]
-----------------------------------------------
```

Task 1.4 Gprof - gmon.out of run after "-p -g" compile 4/4

```
                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]
-----------------------------------------------

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
```

Thank you for your attention!