HPC Lab

Assignment 4

Erkin Kirdan - Manuel Rothenberg - Gabrielle Poerwawinata

1. Superuseful application - GNU gprof

First, we compiled the program just with the added option -pg for the compiler as well as the linker. As in the assignment predicted, we didn't see the Kernel-Functions. We expected to see them, from the information of the documentary of the compiler.
But we also used the optimization level 3 compiler flag, which does several optimization on the code. We figured out, that with this flag, the Kernel-Functions probably get inlined within the main function. That's the reason why they don't show up in gprof, if -O3 is set.

We then removed the -O3 option and added -p (newer flag for deprecated -pg) as well as -g. Here, -p is for compiling and linking for function profiling with gprof and -g is for telling the compiler to generate full debugging information in the object file.

With this options, we made sure, that the compiler doesn't inline the Kernel-functions and we obtained the following output of gprof (see GIT-repository for full output).

```
                     Call graph (explanation follows)


granularity: each sample hit covers 2 byte(s) for 0.03% of 30.22 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]     89.3    0.00   26.99                 main [1]
               21.29    0.00    7972/7972        kernel2(double*, double*, int) [2]
                2.39    3.00    2028/2028        kernel1(double*, double*, int) [3]
                0.31    0.00       1/1           Stopwatch::stop() [9]
                0.00    0.00       1/1           Stopwatch::Stopwatch() [28]
                0.00    0.00       1/1           Stopwatch::start() [25]
-----------------------------------------------
               21.29    0.00    7972/7972        main [1]
[2]     70.5   21.29    0.00    7972          kernel2(double*, double*, int) [2]
-----------------------------------------------
                2.39    3.00    2028/2028        main [1]
[3]     17.8    2.39    3.00    2028          kernel1(double*, double*, int) [3]
                3.00    0.00 2028000000/2028000000     function(double) [4]
-----------------------------------------------
```

So, according to this result, 70% of the total execution time was spent in kernel2, compared to ~18% spent in kernel1. Concluding of this analysis, we should optimize kernel2 first, because it takes much more execution time.

With this configuration we got the Kernel-functions showing up in the report, but we lost the optimization of -O3. So to get both, we should make sure, that the functions don't get inlined, even if we use -O3.

2. Quicksort - Intel VTune Amplier XE

Apart from basic hotspots there is advanced hotspots. Basic Hotspots is software-based sampling with a default resolution of 10 ms whereas Advanced Hotspots is hardware-based sampling with a default resolution of 1 ms.

For the quicksort application, sampling rate is already much larger than the execution time, so its not needed to make an advanced hotspots profiling.

The optimal final clause does depend on the number of threads since number of threads directly determine the spawning resolution. For example, for larger number of threads, final clause should be much lower. On the other hand, the optimal final clause does not depend on the array length.

3. CG - Scalasca

1.)

Since it always took very long to get a job running on CoolMUC2, we tried to install scalasca on CoolMUC3. We got to the point were the following message came up:

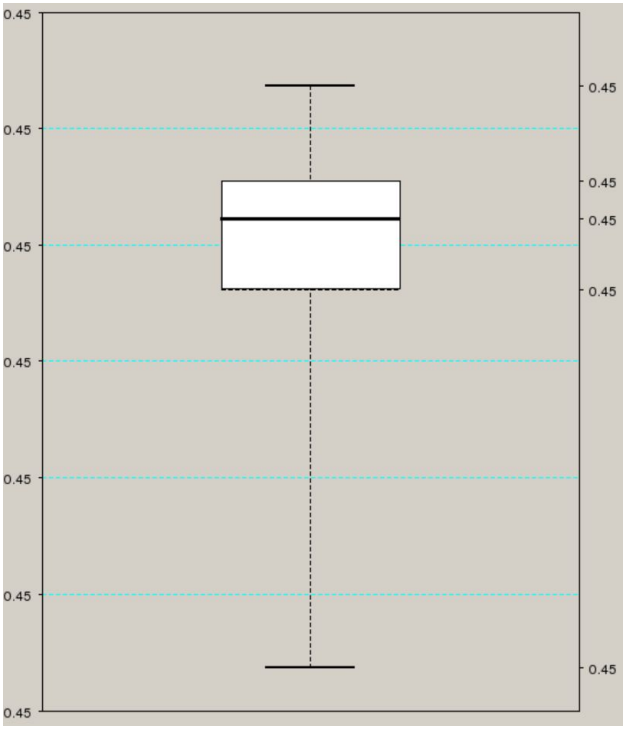
```
di34mip@mpp3-login8:~/code/a4_3_cg> scalasca -instrument --mpp=mpi mpiCC poisson.cpp
Score-P instrumenter "scorep" not found on PATH!
```

So we ran our jobs on CoolMUC2, but weren't able to perform a lot of executions, due to the long waiting time.

2.)

On CoolMUC2 we were able to profile our program from Assignment 3 and detected an odd thing. The screenshots are from a run with 10 tasks and a mesh_width of 0.01.

MPI_Init took roughly 85% of the execution time, where the actual solving took only 9%.



The BoxPlot on the left also shows, that the time used by MPI_Init is nearly exactly evenly distributed to all 10 Tasks.
Every task took about 0.45 seconds for MPI_Init.

3.)

To get just a report with the information we want to see, we included the header `<scorep/SCOREP_User.h>` and added scorep-regions to interesting code areas which we wanted to analyze. We compiled it with the flags `--user --nocompiler` (see poisson_manual_instrumentation.cpp in the GIT-repository for code). Unfortunatelly, we couldn't get any results of this, because CoolMUC2 didn't grant us execution time even after hours of waiting.

4.)

For a hybrid application (OMP/MPI), we took our code from Assignment 3 and added `#pragma omp parallel for` to the outer loops of the functions g_product_operator, g_scale_add, g_scale and g_dot_product (see poisson_with_omp.cpp in the Git-repository for code). To compile the code we added the flag -qopenmp to mpicc. To run the program we used the same configurations like in Assignment 3, but before mpiexec, we exported `OMP_NUM_THREADS=4`. This could also be another number instead of 4, because as read in the documentation every MPI-task should spawn the in OMP_NUM_THREADS specified number of tasks on it's own. These spawned OMP Tasks are then used to parallelize the loops within the specified functions. So the logic of the MPI programming and problem dividing shouldn't change.
Unfortunately we also couldn't test this implementation, because both CoolMUC2 and CoolMUC3 were occupied and didn't grant us execution time even after hours of waiting.
In theory, this implementation should give us a speedup, because the MPI task has more calculation power for the actual calculations. The Communication Overhead will stay the

same, but the calculations should be faster by a factor nearly the same number of OMP_NUM_THREADS.

4. Likwid-perfctr

1. a. In order to use the LIKWID Marker API, we have to include likwid.h. The overview look of source code will be like below:
   We also have to set the LIKWID header that contains a set of macros which allows you to activate the Marker API by defining LIKWID_PERFMON during build of your software. Therefore we need to add header in below to game.cpp source code.

```
#ifdef LIKWID_PERFMON
#include <likwid.h>
#else
#define LIKWID_MARKER_INIT
#define LIKWID_MARKER_THREADINIT
#define LIKWID_MARKER_SWITCH
#define LIKWID_MARKER_REGISTER(regionTag)
#define LIKWID_MARKER_START(regionTag)
#define LIKWID_MARKER_STOP(regionTag)
#define LIKWID_MARKER_CLOSE
#define LIKWID_MARKER_GET(regionTag, nevents, events, time, count)
#endif

LIKWID_MARKER_INIT;
LIKWID_MARKER_THREADINIT;

LIKWID_MARKER_START("Compute");
// Your code to measure
LIKWID_MARKER_STOP("Compute");
LIKWID_MARKER_CLOSE;
```

   We also have to include and library paths for LIKWID. The command line to compile the game.c program is look like below:

```
icpc -DLIKWID_PERFMON -L/lrz/sys/tools/likwid/likwid-4.1/lib/
-I/lrz/sys/tools/likwid/likwid-4.1/include/ game.cpp -o game.out -llikwid
```

   To run application serially, do this command:

```
likwid-perfctr -C 0 -g <event_name> -m ./game.out
```

   where -g is the group of event

   B. likwid-perfctr -a provide us available event group. Event group that we think are relevant:

   - FLOPS_AVX : it includes arithmetic, comparison and data transfer instruction.
   - MEM : diagnosing ccNUMA issues
   - L2 cache bandwidth : measures in socket bandwidth saturation

- L3 cache bandwidth : measured shared cache bandwidth saturation
- Energy: measuring energy and power consumption

C. Our suggestion to improve the performance of the application:

- Reduce the number of arithmetic operation done by the program. It can be done by modifying this part of code:

```
for (int n = 0; n < N; ++n) {
        bodies[n]->move(0.0, 0.0, 0.5 * 9.81 * dt * dt);
 }
```

into

```
 double dz = 0.5 * 9.81 * dt * dt;
 for (int n = 0; n < N; ++n) {
        bodies[n]->move(0.0, 0.0, dz);
 }
```

2. A. Number of flops:
In the dtrmv.cpp there are involved several times of arithmetic operation such as multiplication and addition. Therefore the formula of counting total number of flops are written like below:

Multiplication 2 N(N-1)/2
Addition 2 N(N-1)/2
Total 2N^2-2N

with N = 10000 then total number of flops is (2*10000)^2-(2*10000) = 399980000 flops for every core.

Load imbalance happens when there is waiting / spinning in barrier. Spin time means that one thread is blocking to wait the resource free from another thread. It consumes high cpu time.
If it is imbalance on barriers and you have big number of iterations in a parallel loop much more than worker threads you can try dynamic scheduling to distribute the work dynamically or reduce dependency between iteration.
B. Command line that needed to run measure dtrmv.cpp:

```
icpc -qopenmp -pthread -DLIKWID_PERFMON -L/lrz/sys/tools/likwid/likwid-4.1/lib/
-I/lrz/sys/tools/likwid/likwid-4.1/include/ dtrmv.cpp -o dtrmv.out -llikwid

export OMP_NUM_THREADS=<number_of_threads>

likwid-perfctr -C <range-of-cpu> -g FLOPS_AVX -m ./dtrmv.out
```

On the code we also need to add:

```
#pragma omp parallel
 {
        LIKWID_MARKER_THREADINIT;
```

}

To add thread to the Marker API and initialize access to the performance counters (start daemon or open device files).

Result of the measurement for different numbers of threads:

t=4

	core 0	core 1	core 2	core 3
INSTR_RETIRED_ANY	412K	479K	449K	518K
CPI	1.31	1.009	1.18	0.98

t=8

	core 0	core 1	core 2	core 3	core 4	core 5	core 6	core 7
INSTR_RETIRED_ANY	222k	243k	254k	262k	287k	239k	301k	237k
CPI	1.29	1.24	1.16	1.11	1.01	1.11	0.97	1.19

t=10

	core 0	core 1	core 2	core 3	core 4	core 5	core 6	core 7	core 8	core 9
INSTR_RETIRED_ANY	723k	620k	643k	702k	717k	629k	598k	645k	675k	646k
CPI	1.17	1.3	1.12	1.12	1.09	1.12	1.16	1.24	1.13	1.12

C. Floating Point Operations Executed is often a better indicator. After Intel SandyBridge this event was no longer provided therefore Intel Haswell use an event of AVX_INSTS.ALL which can be called by event FLOPS_AVX. Instruction retired may be a good alternatives of useful workload at least for numerical / FP intensive codes.

D. In order to make dtrmv.cpp load balance, we modified several parts of our code:

```
for (int i = 0; i < N; ++i) {
        double sum = 0.0;
        for (int j = i; j < N; ++j) {
        sum += A[i*N + j] * x[j];
        }
        y[i] = sum;
    }
```

to become

```
for (int i = 0; i < N; ++i) {
        double sum = 0.0;
        for (int j = 0; j < N; ++j) {
        sum += A[i*N + j] * x[j];
        }
        y[i] = sum;
    }
```

The problem before the code was modified is the use of variable x[j]. Therefore we give the full iteration from 0 to N in order to give enough time for calculation until iteration i is incremented.

Result after load balancing:
t=4

	core 0	core 1	core 2	core 3
INSTR_RETIRED_ANY	508k	494k	510k	470k
CPI	1.27	1.25	1.26	1.3

t=8

	core 0	core 1	core 2	core 3	core 4	core 5	core 6	core 7
INSTR_RETIRED_ANY	281k	273k	279k	236k	262k	277k	282k	285k
CPI	1.52	1.55	1.49	1.66	1.54	1.55	1.51	1.50

t=10

	core 0	core 1	core 2	core 3	core 4	core 5	core 6	core 7	core 8	core 9
INSTR_RETIRED_ANY	499k	367k	572k	316k	527k	387k	521k	330k	517k	542k
CPI	1.20	1.86	1.06	1.98	1.07	1.44	1.11	1.9	1.10	1.08