Sheet 03

PS Parallel Programming

Patrick Wintner

March 25, 2025

1 Mandelbrot

The execution time of the program mandelbrot is measured.

1.1 Source Code

```
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// Include that allows to print result as an image
#define STB_IMAGE_WRITE_IMPLEMENTATION
#include "stb_image_write.h"

// Default size of image
#define X 1280
#define Y 720
#define MAX_ITER 10000

void calc_mandelbrot(uint8_t image[Y][X]) {
        for(size_t i=0; i<Y; ++i) {
                for(size_t j=0; j<X; ++j) {
                        double x=0;
                        double y=0;
                        double cx=(double)j/(X-1)*3.5-2.5; // scale j to [-2.5, 1]
```

```
                        double cy=(double)i/(Y-1)*2-1; // scale i to [-1. 1]
                        size_t iteration=0;
                        while(x*x+y*y<2*2 && iteration<MAX_ITER) {
                                double x_tmp=x*x-y*y+cx;
                                y=2*x*y+cy;
                                x=x_tmp;
                                iteration=iteration+1;
                        }
                        char norm_iteration=iteration*255/MAX_ITER; // scale iteration t
                        image[i][j]=norm_iteration;
                }
        }
}

int main() {
        uint8_t image[Y][X];

        calc_mandelbrot(image);

        const int channel_nr = 1, stride_bytes = 0;
        stbi_write_png("mandelbrot.png", X, Y, channel_nr, image, stride_bytes);
        return EXIT_SUCCESS;
}
```

1.2 Measurement Method

The measurement was done on the LCC3 cluster by calling sbatch job.sh. The following scripts are involved in the measurement process.

1.2.1 SLURM Job Script

```
#!/bin/bash

# Execute job in the partition "lva" unless you have special requirements.
#SBATCH --partition=lva
# Name your job to be able to identify it later
#SBATCH --job-name test
# Redirect output stream to this file
#SBATCH --output=output.log
# Maximum number of tasks (=processes) to start in total
```

```
#SBATCH --ntasks=1
# Maximum number of tasks (=processes) to start per node
#SBATCH --ntasks-per-node=1
# Enforce exclusive node allocation, do not share with other jobs
#SBATCH --exclusive

./main.sh
```

1.2.2 Main Script

```
#!/bin/bash
# Usage: ./main.sh
MEASUREMENT_RESULTS=mandelbrot_measurements.log
PROCESSED_RESULTS=mandelbrot_processed.log
make
rm $MEASUREMENT_RESULTS $PROCESSED_RESULTS
for ((i=0;i<=2;++i)); do
        /bin/time -f "%e" -a -o $MEASUREMENT_RESULTS ./mandelbrot
done
./process_results $MEASUREMENT_RESULTS $PROCESSED_RESULTS
make clean
```

The measurement results are stored in mandelbrot_measurments.log, which is read by process_results to compute the average execution time and standard deviation, which are stored in mandelbrot_processed.log.

1.3 Measurement Results

time/s	mean/s	standard deviation/s
17.77	17.74	0.0265
17.72		
17.73		

1.4 Suggestions for performance improvement and parallelisation

The calculations of the colour of different pixels are independent, therefore those calculations could be done parallel.

2 False Sharing

The execution time of two versions of the program false_sharing are compared, followed by a more detailed analysis using perf.

2.1 Source Code

2.1.1 Version 1

```c
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

#define MAX_NUM_THREADS 16

int main(int argc, char **argv) {

    if (argc != 2) {
        printf("Usage: %s <problem_size>\n", argv[0]);
        return EXIT_FAILURE;
    }

    int problem_size = atoll(argv[1]);

    double start, end;
    int* volatile sum = (int*)calloc(MAX_NUM_THREADS, sizeof(int));

    start = omp_get_wtime();

    #pragma omp parallel
    {
        int tid = omp_get_thread_num();
        for (int i = 0; i < problem_size; i++) {
            sum[tid]++;
        }
    }

    end = omp_get_wtime();

    int total_sum = 0;
    for (int i = 0; i < MAX_NUM_THREADS; i++) {
```

```
        total_sum += sum[i];
    }

    printf("Total sum: %d\n", total_sum);
    printf("Time taken: %f seconds\n", end - start);

    free(sum);

    return EXIT_SUCCESS;
}
```

2.1.2 Version 2

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

#define MAX_NUM_THREADS 16
#define FACTOR 16

int main(int argc, char **argv) {

    if (argc != 2) {
        printf("Usage: %s <problem_size>\n", argv[0]);
        return EXIT_FAILURE;
    }

    int problem_size = atoll(argv[1]);

    double start, end;
    int* volatile sum = (int*)calloc(MAX_NUM_THREADS*FACTOR, sizeof(int));

    start = omp_get_wtime();

    #pragma omp parallel
    {
        int tid = omp_get_thread_num();
        for (int i = 0; i < problem_size; i++) {
            sum[tid*FACTOR]++;
        }
    }
```

```
    end = omp_get_wtime();

    int total_sum = 0;
    for (int i = 0; i < MAX_NUM_THREADS; i++) {
        total_sum += sum[i*FACTOR];
    }

    printf("Total sum: %d\n", total_sum);
    printf("Time taken: %f seconds\n", end - start);

    free(sum);

    return EXIT_SUCCESS;
}
```

2.1.3 Differences and Implications

Both versions allocate storage dynamically to a pointer called sum. Each thread is responsible for incrementing exactly one value of the allocated memory (therefore they should not influence each other), until it equals the value given as command line parameter. The final result is the sum of all elements.

In the first version, there is no padding between the memory locations used by the threads for the computation. This means that it is likely that several memory locations used by different threads will be stored in the same cache line. If a thread writes to a memory location in a cache line, it invalidates also all other data stored in the same cache line. Therefore if another thread wants to increment another value stored in the same cache line, the previous thread has to write the cache line back into memory, causing significant delay (even though otherwise the threads are independent and the values are stored in the cache!).

In the second version, there is padding between the memory locations used for incrementing, hopefully preventing that memory locations of different threads are getting loaded into the same cache line.

2.2 Measurement Method

The measurement was done on the LCC3 cluster by calling ./execall. The following scripts are involved in the measurement process.

2.2.1 Execall Script

```
sbatch ./job.sh
sbatch ./job.sh "perf stat"
sbatch ./job.sh "perf stat -e LLC-load-misses -e LLC-store-misses"
```

The first job measures the execution time of both versions with six threads on either one or evenly distributed on two processors. The second job uses perf to get an high-level overview of events happening during execution, while the third looks specifically at cache events.

2.2.2 SLURM Job Script

```
#!/bin/bash

# Execute job in the partition "lva" unless you have special requirements.
#SBATCH --partition=lva
# Name your job to be able to identify it later
#SBATCH --job-name false_sharing
# Redirect output stream to this file
#SBATCH --output=%x_%j.log
# Maximum number of tasks (=processes) to start in total
#SBATCH --ntasks=1
# Maximum number of tasks (=processes) to start per node
#SBATCH --ntasks-per-node=1
# Enforce exclusive node allocation, do not share with other jobs
#SBATCH --exclusive

module load gcc/12.2.0-gcc-8.5.0-p4pe45v
make clean
make
./main.sh $*
make clean
```

2.2.3 Main Script

```
#!/bin/bash
export PROBLEM_SIZE=100000000
export OMP_NUM_THREADS=6
export GOMP_CPU_AFFINITY=0,1,2,3,4,5
```

```
echo "running with "$OMP_NUM_THREADS" Threads on cores "$GOMP_CPU_AFFINITY
$* ./false_sharing $PROBLEM_SIZE
$* ./false_sharing_2 $PROBLEM_SIZE

export GOMP_CPU_AFFINITY=0,1,2,6,7,8
echo "running with "$OMP_NUM_THREADS" Threads on cores "$GOMP_CPU_AFFINITY
$* ./false_sharing $PROBLEM_SIZE
$* ./false_sharing_2 $PROBLEM_SIZE
```

2.3 Measurement Results

2.3.1 execution time

# processors	1	2
$t_{false\_sharing\_1}/s$	3.14e-1	3.91e-1
$t_{false\_sharing\_2}/s$	2.07e-1	2.07e-1

The second version is not affected by increasing the number of processors, while the first suffers an increase in execution time and is generally slower. The reasons why the first version is slower are stated above. Considering that communication between cores of different processors is likely slower than that of cores on the same processor and that there are likely lots of write-back requests when running the first version, it is not surprising that distributing the threads on different processors increases the execution time.

2.3.2 Perf - Overview

The following data was measured with all six threads on one processor.

```
Performance counter stats for './false_sharing 100000000':

          1,812.40 msec task-clock:u              #    5.648 CPUs utilized
                 0      context-switches:u        #    0.000 /sec
                 0      cpu-migrations:u          #    0.000 /sec
                81      page-faults:u             #   44.692 /sec
     5,265,442,510      cycles:u                  #    2.905 GHz                      (83.39%)
     4,072,468,552      stalled-cycles-frontend:u #   77.34% frontend cycles idle     (83.18%)
       947,310,743      stalled-cycles-backend:u  #   17.99% backend cycles idle      (66.64%)
     2,413,081,608      instructions:u            #    0.46  insn per cycle
                                                  #    1.69  stalled cycles per insn  (83.40%)
       604,385,457      branches:u                #  333.473 M/sec                    (83.45%)
             7,127      branch-misses:u           #    0.00% of all branches          (83.44%)
```

```
       0.320876265 seconds time elapsed

       1.796202000 seconds user
       0.001986000 seconds sys

Performance counter stats for './false_sharing_2 100000000':

          1,250.96 msec task-clock:u              #    5.817 CPUs utilized
                 0      context-switches:u        #    0.000 /sec
                 0      cpu-migrations:u          #    0.000 /sec
                80      page-faults:u             #   63.951 /sec
     3,631,302,842      cycles:u                  #    2.903 GHz                      (83.29%)
     2,439,312,866      stalled-cycles-frontend:u #   67.17% frontend cycles idle     (83.22%)
       593,658,873      stalled-cycles-backend:u  #   16.35% backend cycles idle      (66.43%)
     2,411,063,345      instructions:u            #    0.66  insn per cycle
                                                  #    1.01  stalled cycles per insn  (83.28%)
       603,202,626      branches:u                #  482.193 M/sec                    (83.48%)
               435      branch-misses:u           #    0.00% of all branches          (83.58%)

       0.215052654 seconds time elapsed
```

The following data was measured with six threads distributed over two processors.

```
Performance counter stats for './false_sharing 100000000':

          2,217.21 msec task-clock:u              #    5.460 CPUs utilized
                 0      context-switches:u        #    0.000 /sec
                 0      cpu-migrations:u          #    0.000 /sec
                81      page-faults:u             #   36.532 /sec
     6,444,167,778      cycles:u                  #    2.906 GHz                      (83.29%)
     5,253,540,958      stalled-cycles-frontend:u #   81.52% frontend cycles idle     (83.31%)
     1,537,181,968      stalled-cycles-backend:u  #   23.85% backend cycles idle      (66.72%)
     2,414,069,616      instructions:u            #    0.37  insn per cycle
                                                  #    2.18  stalled cycles per insn  (83.36%)
       603,408,765      branches:u                #  272.148 M/sec                    (83.36%)
             6,968      branch-misses:u           #    0.00% of all branches          (83.45%)

       0.406083446 seconds time elapsed

Performance counter stats for './false_sharing_2 100000000':

          1,242.99 msec task-clock:u              #    5.881 CPUs utilized
                 0      context-switches:u        #    0.000 /sec
                 0      cpu-migrations:u          #    0.000 /sec
                81      page-faults:u             #   65.165 /sec
     3,608,890,769      cycles:u                  #    2.903 GHz                      (83.11%)
     2,421,206,931      stalled-cycles-frontend:u #   67.09% frontend cycles idle     (83.11%)
       635,247,518      stalled-cycles-backend:u  #   17.60% backend cycles idle      (67.09%)
     2,402,511,506      instructions:u            #    0.67  insn per cycle
                                                  #    1.01  stalled cycles per insn  (83.59%)
       600,892,351      branches:u                #  483.424 M/sec                    (83.59%)
```

```
               400      branch-misses:u           #    0.00% of all branches          (83.19%)

       0.211362765 seconds time elapsed
```

The number of processors does not have a significant impact on the second version. The number of clock cycles and branch misses of the first version are significant higher than those of the second version. Increasing the number of processors increases the number of clock cycles for the first version.

2.3.3 Perf - Details

The following data was measured with all six threads on one processor.

```
Performance counter stats for './false_sharing 100000000':

               507      LLC-load-misses:u
               163      LLC-store-misses:u

       0.318045312 seconds time elapsed

       1.791884000 seconds user
       0.000994000 seconds sys

 Performance counter stats for './false_sharing_2 100000000':

               146      LLC-load-misses:u
                58      LLC-store-misses:u

       0.212332630 seconds time elapsed

       1.232476000 seconds user
       0.002983000 seconds sys
```

The following data was measured with six threads distributed over two processors.

```
Performance counter stats for './false_sharing 100000000':

         1,153,111      LLC-load-misses:u
         1,452,548      LLC-store-misses:u

       0.402423131 seconds time elapsed

       2.198254000 seconds user
       0.001989000 seconds sys

Performance counter stats for './false_sharing_2 100000000':
```

```
          250      LLC-load-misses:u
           86      LLC-store-misses:u

  0.209930705 seconds time elapsed

  1.235174000 seconds user
  0.001980000 seconds sys
```

The number of LLC-load-misses of the first version is signifiantly higher, especially when the threads are distributed over two processors.