Università della Svizzera italiana

Facoltà di scienze informatiche

High Performance Computing **2017**

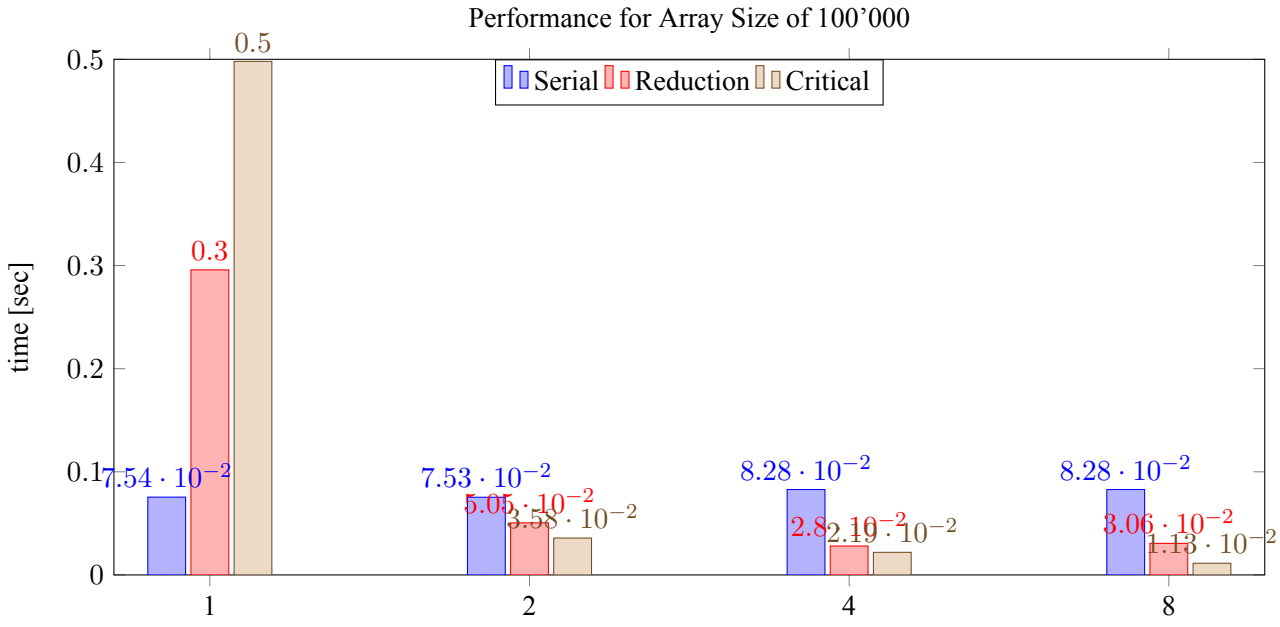
Student: Timon Willi

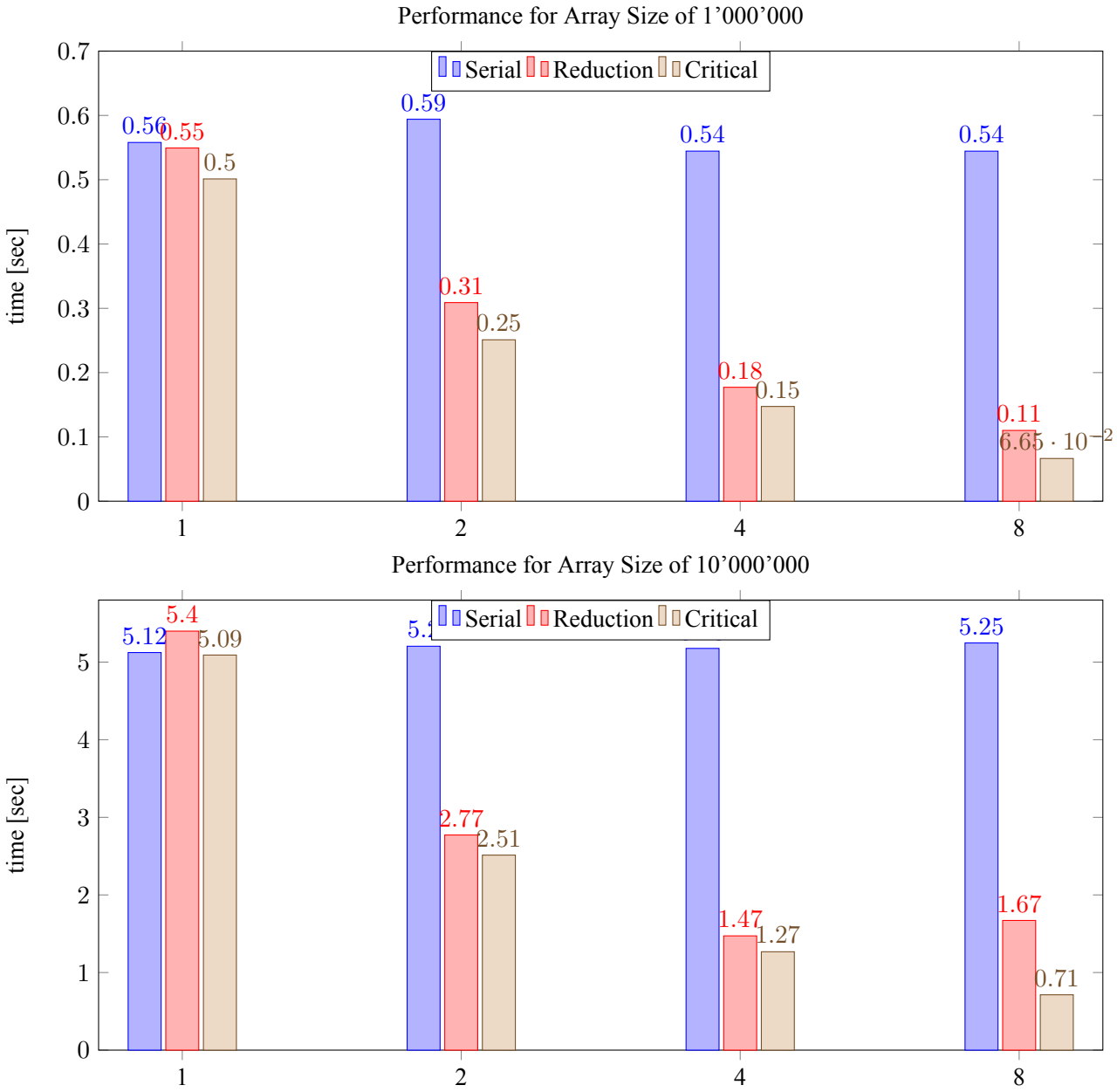
Solution for Assignment 3

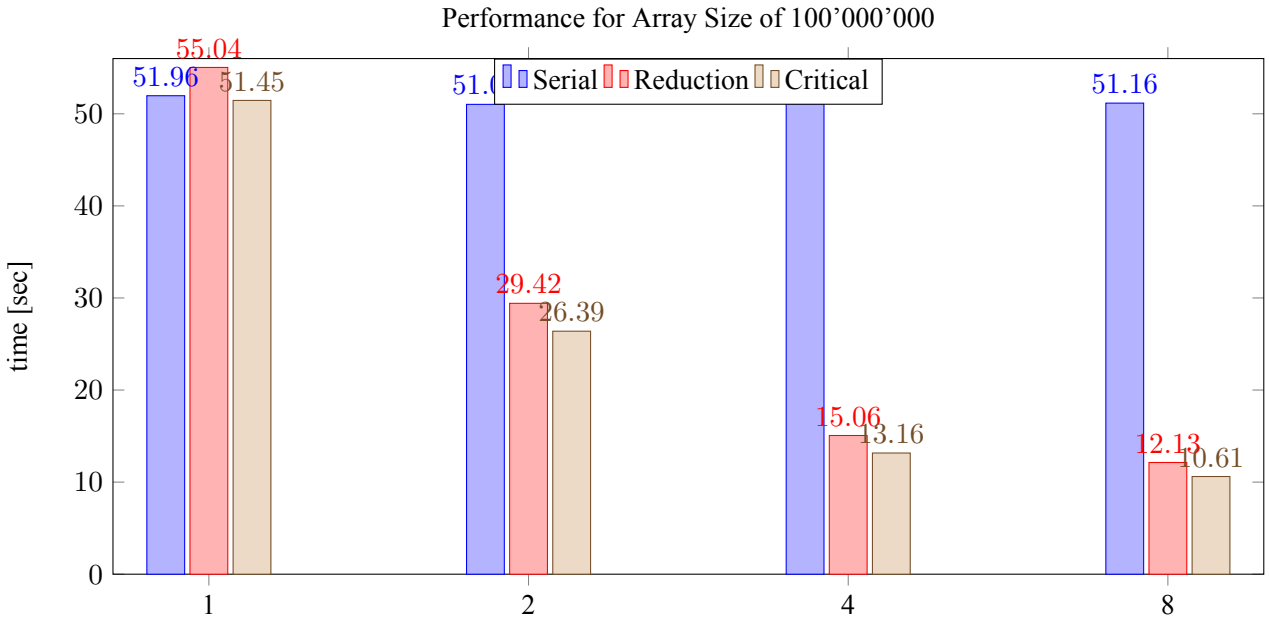
Due date: 30. October 2017, 8:00am

1. Parallel reduction operations using OpenMP *(30 Points)*

1.1. Parallel Performance

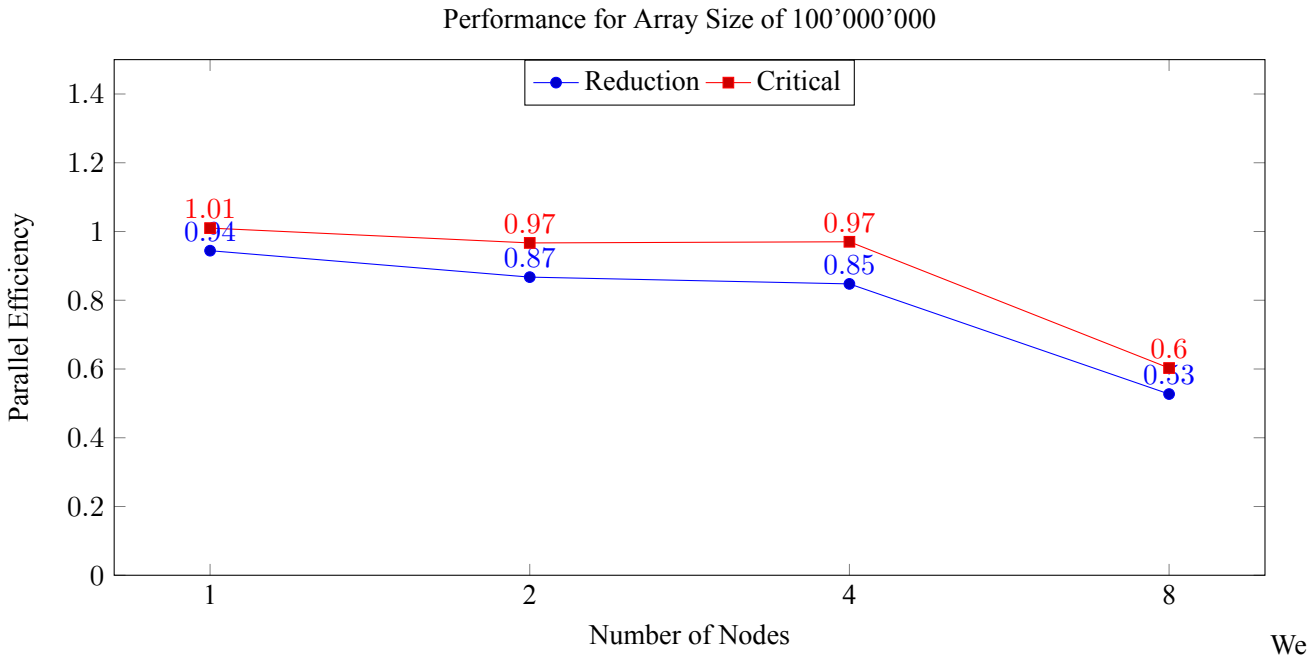



Performance for Array Size of 1’000’000
time [sec]
Serial
Reduction
Critical
0.56
0.55
0.5
0.59
0.31
0.25
0.54
0.18
0.15
0.54
0.11
$6.65 \cdot 10^{-2}$
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
1
2
4
8
Performance for Array Size of 10’000’000
time [sec]
Serial
Reduction
Critical
5.12
5.4
5.09
2.77
2.51
1.47
1.27
5.25
1.67
0.71
0
1
2
3
4
5
1
2
4
8



As one can see from the performance graphs, on average the reduction method performs better than the serial calculation and the critical version performs better than the reduction method. One other observation that can be made is that the higher the number of threads, the lower the time it takes to finish. This of course only when the algorithm allows for parallelization. We can see that in the case of only one thread, the serial version performs better than the other methods when the array is small. This is probably due to the overhead coming with OpenMP. This effect however diminishes, the bigger the array gets.

1.2. Parallel Efficiency



We see that the efficiency is not linear with the number of additional nodes, since it decreases. Optimal would

have been if it stayed at 1 or would even increase. The parallel Efficiency was calculated using the formula $E_{red} = \frac{1}{N} * \frac{Sequential(N)}{Reduction(N)}$ and $E_{crit} = \frac{1}{N} * \frac{Sequential(N)}{Critical(N)}$ respectively. We used the largest array size for the comparison of speeds, since this minimizes the effect of the overhead.

2. Visualizing the Mandelbrot Set *(30 Points)*

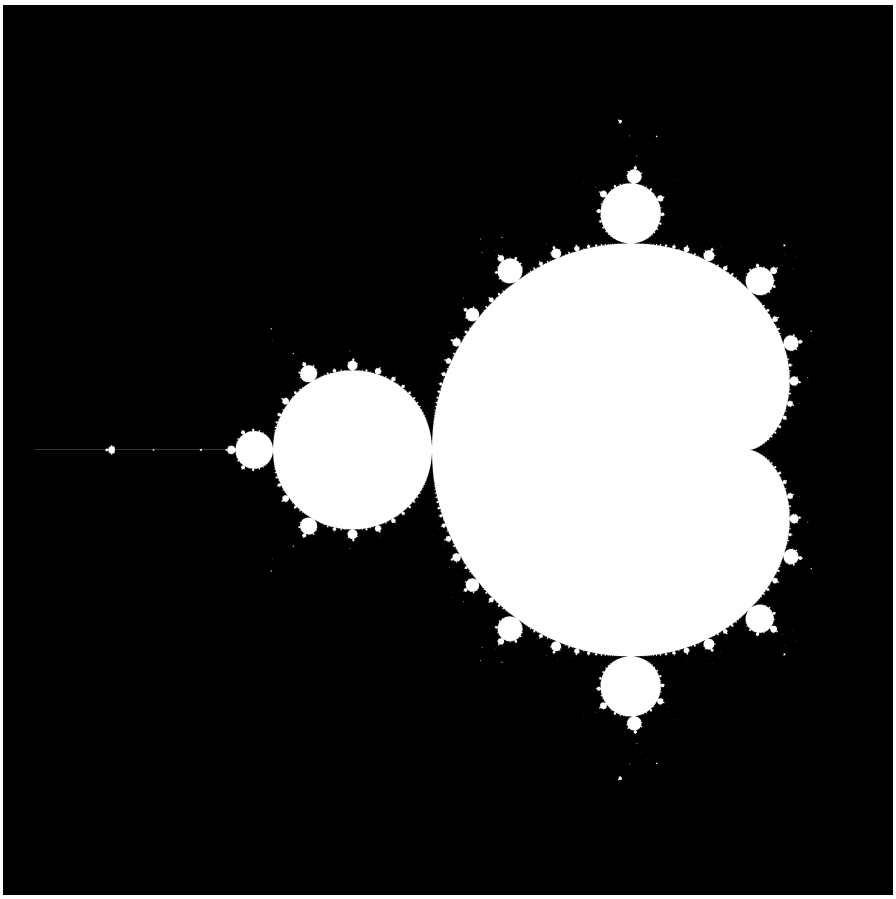


Figure 1. Mandel Picture 4096x4096

As one can see in Table 1 and Table 2, the time was nearly divided by 5 for all picture sizes, as well as the time spent per pixel and average time per iteration. This lead to more Flops per second.

3. Parallel Mandelbrot *(20 Points)*

4. Bug Hunt *(20 Points)*

4.1. Bug 1

This code contains faulty pragma omp statements. One should not use the ”for” statement if the pragma statement is not immediately before a for-loop. We thus should create a separate pragma omp for statement right before the

Table 1. Sequential Performance

Total Time [ms]	474036	118528 ms	34605.1	8626.14
Image Size [Pixels]	4096^2 = 16777216	2048^2 = 4194304	1024^2 = 1048576	512^2 = 262144
Avf. Time per pixel [µs]	28.2548	28.2593	33.002	32.9061
Avg. time per iteration [µs]	0.00417195	0.00417253	0.00487094	0.00485861
Iteration/second	2.39696e+08	2.39663e+08	2.05299e+08	2.0582e+08
MFlop/s	1917.57	1917.3	1642.39	1646.56

Table 2. Parallel Performance, OMP_NUM_THREADS=16 (default)

Total Time [ms]	94430.7	24582.8	5979.66	1760.81
Image Size [Pixels]	4096^2 = 16777216	2048^2 = 4194304	1024^2 = 1048576	512^2 = 262144
Avf. Time per pixel [µs]	5.62851	5.86098	5.70265	6.71696
Avg. time per iteration [µs]	0.000830873	0.000864958	0.000840842	0.00098984
Iteration/second	1.20355e+09	1.15613e+09	1.18928e+09	1.01026e+09
MFlop/s	9628.42	9249.01	9514.28	8082.11

loop. Additionally, the ”schedule” statement belongs to the for-loop, we thus have to move it to the pragma omp for statement.

4.2. Bug 2

First of all, the for-loop which is calculating the variable ”total”, is creating a racing condition, since the variable total is accessed by different threads at the same time and re-used for their respective calculation. This can be solved either with the reduction statement or by creating private local variables and then by adding them together in a critical section. Also the variable tid needs to be set to private, otherwise it will be overwritten by the last thread overwriting it and only this tid will be printed.

4.3. Bug 3

The barrier in the print_results function waits for all threads. However, only two threads will pass that barrier, since a section is only executed by one thread respectively. Thus the program will never end since the threads will wait forever, while the other threads wait at the other barrier.

4.4. Bug 4

The program is accessing a point in memory that does not exist, since there is a stack overflow. This occurs because for each thread, the private values are put on a stack with limited memory. If the array is too big, it will not fit on the stack. Since we have 1048 elements that each take 8 bytes, we have 8384 in total. However, our stack size is only 8192 bytes big, leading to an overflow.

4.5. Bug 5

There is a deadlock because section 1 is asking for a lock on b and section 2 is asking for a lock on a. They will not continue until they receive their lock. However section 1 cannot release until it has the lock for b and section 2 cannot release until it has the lock for a. Thus a deadlock. With the omp_test_lock with can implement a non-blocking while-loop that tests the avaiable of the lock. If a section is finished with its first loop, it releases the lock and checks if both locks are available. If they are, it gets the locks and continues, while the other section is waiting in its while-loop.