UNIVERSITY OF AMSTERDAM

Parallelization with PThreads

Exploring heat dissipation, sort problem and mutual exclusion concepts

Kian Paimani / 11622849
Jonas Theis / 12142964

Universiteit van Amsterdam
Vrije Universiteit Amsterdam
Course: Programming Multi-core and Many-core Systems
Course instructor: dr. Clemens Grelck

20th August 2018

1 Introduction

In this report we will demonstrate our findings from experimenting with PThreads and different concepts of mutual exclusion in multiple problems. In this assignment, our main concern is multi-thread/core parallelization. For this reason, before explaining the results, we will briefly discuss the available CPUs on DAS worker nodes and examine their ability for multi-core execution.

As opposed to OpenMP[1], which was discussed in the second assignment, *PThreads* propose a low-level programming model for multi-cores. In other words, whilst in OpenMP most of the burden was being carried by the compiler, in PThreads all of that responsibility is programmer's duty. In essence, PThreads give **full control** over the behavior of each spawned thread. During the course of this report, we will recall some of these flexibilities.

To further elaborate on the concepts of PThreads, three different types of experiments have been conducted, namely:

- **Heat Dissipation**: An iterative matrix cell update process based on neighbouring cells.
- **PipeSort**: A simple, one thread per element, sorting algorithm that requires a great deal of synchronization among huge numbers of threads.
- **Histogram**: One-shot iteration over a grayscale image, represented as 2 dimensional array, to compute its histogram.

For the rest of this report, in section 2 we will explain the implementation details of each of these tasks. In section 3 we will evaluate the performance of each implementation and compute their speedup based on a specific baseline. The baseline for each part is, of course, different and it is explained in the respective section. Finally, in section 4 we conclude about PThreads and the explored concepts of mutual exclusion.

2 Implementation Details

In this section we will describe implementation details of all three parts of this assignment. We will focus on the high level concepts and omit negligible details.

2.1 Heat Dissipation

In order to implement the same heat dissipation problem using PThreads, we strive to manually apply all steps that we assume the OpenMP compiler is taking care of. Hence, we will try to map the implementation details of our PThreads version to an OpenMP feature. The most important of these are:

- **Data Distribution**
- **Parallel Section**
- **Thread Private/Shared Data**
- **Synchronization**

We will now explore each of these points.

The **Data Distribution** of the PThreads program has to be applied manually. For this, we have made reasonable decision to *not* duplicate any data and pass the entire matrix pointer to each thread as a shared variable. Hence, the important aspect becomes dividing the iteration space among threads. Needless to say, for many reasons, most importantly cache efficiency, dividing the matrix row-wise among threads is the optimal solution. For this, we have used a comprehensive and more importantly **fair** pre-processing procedure to split the row indexes among threads. To achieve this, first *integer division* of `NUM_ROWS / NUM_THREADS` is calculated. By this point, we know that each thread will at least get this amount of rows. For the remaining part, we will assign the remaining rows *one by one* to all threads sequentially until they are finished. This ensures the highest possible degree of fairness among thread and prevents **load imbalance** as much as possible. Fairness cause synchronization of threads to become cheaper.

As an example, the following program log indicates the index division of a 1000 by 1000 matrix among 8 threads.

```
[Thread 0] :: 1 -> 126 [copy: 0 -> 126][weigth=125]
[Thread 1] :: 126 -> 251 [copy: 126 -> 251][weigth=125]
[Thread 2] :: 251 -> 376 [copy: 251 -> 376][weigth=125]
[Thread 3] :: 376 -> 501 [copy: 376 -> 501][weigth=125]
[Thread 4] :: 501 -> 626 [copy: 501 -> 626][weigth=125]
[Thread 5] :: 626 -> 751 [copy: 626 -> 751][weigth=125]
[Thread 6] :: 751 -> 876 [copy: 751 -> 876][weigth=125]
[Thread 7] :: 876 -> 1001 [copy: 876 -> 1002][weigth=125]

// all threads compute with the following index boundaries:
for (int i = start_idx; i < end_idx; i++) {
   for (int j = 0; j < WIDHT; j++) {/* ... */}
}
```

As seen in the log, another parameter is also being calculated as `copy index`. This value indicates that at the end of each iteration, each thread should copy the first and last column of which rows. For this reason, the copy index of first and last thread is different because they also have to copy the halo rows.

This precise division with **no overlap** allows threads to share the same pointer for both read and write to cells without the need to use any synchronization primitive (such as mutex *etc.*).

Having a suitable division of data in hand, the next dilemma is about regions to choose as **Parallel Region**. One of the possible negative aspects of OpenMP is that by restricting parallelism to high level pragmas, it is relatively hard to have a long parallel region. Albeit the runtime can be smart enough to compensate this[1]. Using PThreads, this issue can be easily eliminated. For our heat implementation, we have used the longest possible parallel region in which almost all operations happen in parallel. Aside from the cell update procedure, the pre/post processings also happen in parallel. Examples of this can be:

- Mirroring first and last column. the `copy_idx` parameters are used for this purpose. This happens at the beginning of each iteration and all threads synchronize after it.
- Computing diff. Each thread will compute the local diff of its own computation range and stores it in a specific index of a shared `diff_buffer`. Then, they all hit a barrier to ensure that all threads are finished with the iteration. Afterwards, each of them will reduce the `diff_buffer` to obtain the maximum diff and decide based on that.

The only *single-threaded* operation is printing intermediate reports in which the first thread computes the statistics and all threads must wait for it to finish. The following snippet shows the gist of the parallel region:

```
for (iter = 1; iter <= maxiter; ++iter) {
       { void *tmp = src; src = dst; dst = tmp; }

       /* Mirror last and first column */
       for (i = params->copy_start_idx; i < params->copy_end_idx ; i++) {/* ... */}
       pthread_barrier_wait(&barrier);
       double maxdiff = 0.0;

       /* compute */
       for (i = start_idx; i < end_idx ; ++i) {
           for (j = 1; j < w - 1; ++j) {/* ... */ }
       }

       /* write local max diff */
       params->diff_buffer[params->id] = maxdiff;

       /* wait for all threads to write local max diff */
       pthread_barrier_wait(&barrier);

       /* compute global max diff */
       double global_diff = 0.0;
       for (i = 0; i < *params->num_threads; i++) {/* ... */}
```

[1] For example, by re-using already existing threads instead of iteratively killing them.

```
        if (global_diff < threshold) {
            break;
        }

        /* thread 0 will print if needed */
        if ( printreport ) {
            if (params->id == 0) {
                if ( iter % period == 0 ) { */ ... */ }
            }
        }
}
```

Regarding **Thread Private/Shared Data**, as mentioned, most of the data used by threads is shared pointers. To summarize all of them, the following struct shows the definition of the parameters passed to each thread:

```
typedef struct thread_params{
    int start_idx;                  // computation index
    int end_idx;                    // computation index
    int copy_start_idx;             // copy index
    int copy_end_idx;               // copy index
    int id;                         // thread id
    int *num_threads;               // total number of threads. Shared pointer.
    double *diff_buffer;            // shared buffer for storing diffs
    size_t *iter;       // shared empty pointer for outputting the number of iterations.
    double ***src_ptr;              // shared pointer to matrix
    double ***dst_ptr;              // shared pointer to matrix
    double ***c_ptr;                // shared pointer to matrix
    struct results* results_ptr;    // shared pointer to result struct
    struct parameters* parameters_ptr; // shared pointer to parameters struct
    struct timeval *before;         // execution start time
} thread_params;
```

One small point worth mentioning is that wherever the data could be re-used, a pointer is used instead of a direct value to save space on stack. While the array pointers are obvious in this manner, using pointers for passing parameter structs or even iteration count is also for the same purpose. Among the parameters, `diff_buffer` serves a special purpose and was discussed in the *Parallel Regions* description.

Lastly, it is also worth mentioning that since the parallel region is relatively large, pre-computing items such as `thread_copy_start_idx` is more efficient that having multiple `if() {...}` clauses in the code.

As already seen, unlike OpenMP where the bare `for loop` of cell update is parallelized and it had an implicit **Synchronization barrier**, for our implementation we have used explicit PThreads

library functions to synchronize the threads.

2.2 PipeSort

The assignment's pipeline sort algorithm works as follows: A *generator thread* produces a sequence of numbers (followed by 2 end signals) and passes every number to a *comparator thread*, which either stores this number if it's the first received number or compares it to its stored number and passes the smaller number to its successor comparator thread. This pattern of comparator threads comparing and passing on is repeated until the end signal is received. Then a comparator thread forwards the end signal followed by its stored number and all following received numbers until the second end signal is received and the thread terminates. The last comparator thread creates an *output thread* which prints all numbers to standard out. There is a bounded buffer between each thread and its successor of a given size.

In our implementation all threads are created in a detached state and the main thread waits on a semaphore, which gets flipped by the output thread when the sorting is done. Thus, the main thread creates the generator thread in a fire-and-forget way and then just waits. The generator creates random numbers and passes them according to the *Producer-Consumer Problem* to its comparator thread.

```
// send number by number into pipeline
for(int i = 0; i < length; i++) {
  sem_wait(empty);
  buffer[next_in] = rand();
  next_in = (next_in + 1) % buffer_size;
  sem_post(full);
}
```

The buffer between a thread and its successor and the needed semaphores to synchronize between the threads according to the *Producer-Consumer Problem* are created on the fly within each thread and passed on to its successor via the struct `Comparator_params`. The accrued resources are freed as soon as both the current thread and its predecessor are done and are ready to terminate.

```
typedef struct Comparator_params {
    int *buffer;
    sem_t *full;
    sem_t *empty;
} Comparator_params;
```

A comparator thread acts as a consumer (consumes number from its predecessor) and producer (forwards numbers to its successor) at the same time. Furthermore, it constitutes a state machine with the following states:

1. `INITIAL` No number received yet. First number received is stored. Next state: `COMPARE_NO_THREAD`
2. `COMPARE_NO_THREAD` There is no next thread yet. Depending on the received number there needs to be a comparator thread (next state: `COMPARE`) or if an END symbol is received an output thread created (next state: `END`). The received number/END symbol is then forwarded to the thread.
3. `COMPARE` All numbers received are compared to the stored number and the lower one is sent to the next comparator thread. If an END symbol is received the end symbol and stored number are forwarded (next state: `END`).
4. `END` All numbers are forwarded to the next thread including the second END signal.Then the thread terminates and frees the shared resources between its predecessor and itself.

At the beginning a comparator thread extracts and stores the buffer and the semaphores from its predecessor thread. Then it creates buffer and semaphores to pass them on to its successor. The main logic is in an infinite while loop, where a number is received in the beginning and then action is taken depending in which state the comparator is.

```
// assign read buffer and semaphores
[...]
// assign write buffer and semaphores
[...]

while(1) {
       sem_wait(read_full);
       read_val = read_buffer[read_out];
       read_out = (read_out + 1) % buffer_size;
       sem_post(read_empty);

       if (state == INITIAL) {
           // store number
           [...]
       } else if(state == COMPARE_NO_THREAD) {
        // create next thread depending on the received number and forward number/END
            signal
           [...]
       } else if(state == COMPARE) {
           // compare numbers and send lower to next thread
           [...]
       } else if (state == END) {
           // forward all numbers and break out of loop upon second end signal
           [...]
       }
   }
```

```
// free resources
[...]
```

The output thread acts as a consumer, which prints all numbers to standard out and checks whether the order in which the numbers arrive is correct (ascending). After all numbers are received the thread posts to the semaphore on which the main thread is waiting on and terminates. The main thread then calculates the total time and the whole program terminates.

In an earlier version, while implementing the PipeSort algorithm we faced a strange behaviour of the program:

- Sometimes numbers were lost, e.g. 20 sent by generator and only 15 received by the output thread
- Sometimes the program got stuck
- Sometimes there were issues when accessing memory
- Sometimes everything worked as expected

After debugging the code we found out, that some of the semaphores created by earlier threads were reused by later threads, but with no direct connection. The appearance of these issues was also arbitrary. Finally, we came to the conclusion that creating semaphores on the stack within a thread and then handing over a pointer to it to the next thread was the issue. Therefore, we create the semaphores on the heap and pass a reference to it to the next thread, which solves all of the previously mentioned problems, because both threads can access elements allocated on the heap independently of the current state of a thread (active, zombie, terminated).

```
// earlier with issues: create a semaphore on the stack
sem_t write_full;

// now: create a semaphore on the heap
sem_t *write_full = malloc(sizeof(sem_t));
```

2.3 Histogram

A histogram of a grayscale image is essentially counting the number of occurrences of each pixel value in the image. Hence, for this assignment where each image is defined as a `NxM` matrix of values in range `[0, 255]`, the result of the computation is a vector of 255 elements where element `i` indicates how many times the pixel value `i` exists in the image. Keeping these information in mind, two ways can be proposed to parallelize the application:

- Parallelize based on **image size**. Each thread will do the counting on a portion of the image. The result could be written either to a shared buffer or a local buffer that needs to be reduced to one single vector at the end. In case of shared buffer, synchronization is needed to avoid race conditions.
- Parallelize based on **image value**. Each thread will iterate the entire image and only count the occurrences of a certain range of pixel values. No synchronization is needed in this version since parallel writes happen to different indexes of the result buffer.

Since the purpose of this assignment is to test the performance of different synchronization mechanisms, we have mostly used the first approach to be able to demonstrate this. To implement a parallel version that uses no synchronization barrier at all, we have implemented both approaches with different tricks to avoid using synchronizations. Nonetheless, we have seen that the *image-value-parallel* version is not efficient at all. This is reasonable, because it practically does not add any speedup to the execution. Instead, it creates more work to be done by forcing each thread to iterate through the entire image. Even worse is that each thread in each iteration must check the value of the pixel and see if it is in its respective range. Hence, we have used a simpler, yet more efficient approach; The work is divided by the rows of the image and each thread will compute and store a local histogram. At the very end, all histograms are merged by the main process to reach a final one.

In all implementations, the image is synthesized by creating a ranged random 2-dim matrix. Furthermore, all executions will also compute the histogram sequentially to ensure the veracity of the result[2]. The reader is encouraged to disable any of the synchronization mechanisms in the code and observe how the result will become invalid in this case. We will now focus on the common portion of all implementations.

The following struct defines the parameters passed to each thread:

```
typedef struct thread_args {
    unsigned int start_idx;
    unsigned int end_idx;
    unsigned int width;
    unsigned int height;
    unsigned int ***restrict img;
    unsigned int **restrict histo;
} thread_args;
```

The `start_idx / end_idx` are used to define the range of rows to calculate in the *image-size-parallel* and the range of pixel values in the *image-value-parallel* approach. Consequently, the update loop of the first approach, splitting based on dimension, is as follows:

```
for (i = start_idx; i < end_idx; i++) {
```

[2] only by using `-s` flag.

```
    for (j = 0; j < WIDTH; j++) {
        (*histo)[(*img)[i][j]]++;
    }
}
```

As seen in the snippet, the outer for loop is bounded to an index and is being parallelized. And for the *image-value-parallel* approach, using pixel value ranges to parallelize:

```
for (i = 0; i < HEIGHT; i++) {
    for (j = 0; j < WIDTH; j++) {
        unsigned int pix_val = (*img)[i][j];
        if (pix_val < end_idx && start_idx >= lb) {
            (*histo)[pix_val]++;
        }
    }
}
```

The difference between the two is self-explanatory. Using this common structure, the only variable part is to restrict access to `(*histo)[(*img)[i][j]]++;` by different synchronization barriers. Note that the *image-value-parallel* approach **does not** need any synchronizations because each thread will only access a dedicated portion of the buffer.

The following snippet shows the gist of some of these mechanism:

```
for (i = 0; i < HEIGHT; i++) {
    for (j = 0; j < WIDTH; j++) {
       pthread_mutex_lock(&mutex);
        (*histo)[(*img)[i][j]]++;
        pthread_mutex_unlock(&mutex);
        /* ------- or ------ */
        __sync_fetch_and_add(&((*histo)[(*img)[i][j]]), 1);
        /* ------- or ------ */
        sem_wait(&sem);
        (*histo)[(*img)[i][j]]++;
        sem_post(&sem);
        /* ------- or ------ */
        __transaction_atomic { (*histo)[(*img)[i][j]]++; }
    }
}
```

Both `__transaction_atomic` and `__sync_fetch_and_add` are special gcc operations[2, 3] that perform a transaction (aka. a block of code) and a single operation in an *atomic* manner.

Before diving into the details of the experiments, some other possible optimization applied should not be overlooked. We have already explained that for an implementation that uses no

synchronization, splitting the task based on rows and merging them at the end is deemed to be faster. Furthermore, in the above snippets, specifically for mutex and semaphore, we observe that we are using a common lock for the entire data structure. While this is safe, it is not reasonable. It is not efficient to block a thread that requests to write and index 1 when another thread wants to write at index 20. To solve this, in the real implementation, an array of mutexes/semaphores are used. With this fine granularity of locking, we ensure that a thread is never blocked without a reason.

3 Evaluation

This section first describes the experimental setup, the procedure of taking measurements and our expectations. Then, the results using the presented setup are displayed and analyzed.

3.1 Experimental setup and expectations

All the experiments were conducted on the DAS4 VU cluster. Each node consists of a dual quad-core, 2x hyperthreading processor with 2.4GHz, 24 GB memory and 2*1TB HDDs interconnected with Gigabit Ethernet and InfiniBand. Measured was only the time of the actual computation without eventual initial time for reading and preprocessing. Furthermore, for the heat dissipation floating point operations per second (FLOPs) are calculated based on the measured time and the size of the problem. Measurement results marked as sequential, are truly conducted as sequential executions of the corresponding program, which means that there is no overhead for creating or managing threads. All applications were compiled using `GCC 6.2.0` and the optimization flag `-O2`.

As mentioned before, a node on the DAS4 consists out of a dual quad-core processor with 2x hardware threading. Therefore, there are 8 cores and 16 hardware threads in total available. It has to be noted, that there is only one floating precision entity per core. This in turn means that effectively only one hardware thread can be used for floating point intense operations.

Based on these hardware specifications of the DAS our expectations for the different applications are as follows:

- **Heat Dissipation**: Assuming that the OMP compiler hints are as optimized as possible, which is not far from truth, we expect a manual PThread version to be *in most cases* almost as good as the OpenMP implementation. Nonetheless, as seen in the dedicated report on OMP benchmarks, anomalies can occur and we will explain them in the next section. Similar to the previous report, we expect the best outputs to be produced with 8 cores/threads.
- **PipeSort**: We expect this sorting algorithm, effectively spawning $N + 2$ threads where N is the number of elements to sort, besides the main thread, will be unreliable in terms of execution time and inefficient. Furthermore, we expect the size of the buffer to affect the

Table 1: PThread results with rectangular input size

	100x100				*1000x1000*				*5000x5000*			
Threads	***Time(S)***	***GFLOPS***	***Speedup***	***PTh/OMP***	***Time(s)***	***GFLOPS***	***Speedup***	***PTh/OMP***	***Time(s)***	***GFLOPS***	***Speedup***	***PTh/OMP***
seq	0.011		-	-	1.195	2.03	-	-	30.28	2	-	-
1	0.012	2.09	1.004	-	1.204	2.01	0.993	-	29.864	2.03	1.014	-
2	0.008	3.02	1.448	0.783	0.628	3.86	1.905	0.946	15.648	3.87	1.936	0.995
4	0.007	3.24	1.552	0.590	0.312	7.76	3.834	1.028	8.578	7.05	3.531	0.975
8	0.010	2.53	1.137	0.350	0.189	12.8	6.332	1.075	8.137	7.43	3.722	0.837
16	0.026	.942	0.452	0.213	0.224	10.8	5.328	0.998	6.448	9.38	4.697	1.123
24	0.049	.495	0.237	1.894	0.164	14.7	7.272	1.954	8.799	6.88	3.442	0.848
32	0.065	.373	0.179	0.342	0.263	9.20	4.544	0.998	7.742	7.81	3.913	0.930

runtime of the program. The bigger the buffer, the faster the execution, because more elements can be processed within one thread before a context switch has to occur. For example if the buffer size is 1, only 1 number can be processed within one thread and then it starts blocking and has to wait for its predecessor. If the buffer size is 4, there possibly could be 4 numbers at once processed.

- **Histogram**: While the histogram application itself can be deemed as an *embarrassingly parallel task*, the high data dependency of its nature, if implemented as explained in section 2.3, can be a performance kill. Hence, we form the following set of hypothesis:
  - If the program is implemented in a way that all threads are compelled to use a shared buffer to store the updates after each pixel check, we do **NOT** expect the performance to improve at all. This is mainly due to the extensive use of synchronization primitives.
  - Nonetheless, just to prove our point, we will include a small sub-test in which no synchronization barrier is used and the threads store a local histogram that will be reduced at the end. We expect this implementation to be way faster.
  - With respect to different synchronization mechanism, we will empirically find out which one is the fastest and argue that based on our knowledge.

3.2 Results: Heat Dissipation

In order to be in sync with the OpenMP result and be able to compare PThread's results with it accurately, we will perform our tests with exactly the same dimension sizes. Consequently, we will start with rectangular input sizes in Table 1. The Speedup column of the table is relative to the sequential implementation. Furthermore, the *PTh/OMP* column demonstrates the relative speedup of the pthread implementation with respect to that of OMP. In other words, it is equal to S_{pth}/S_{omp}.

It can be seen in the table that the results for 100x100 input size are not a good representative. In fact, they include some of the most unusual anomalies of our gathered data. While in most cases the PThread version is performing worse than the OMP version, we observe that with 24 cores the performance of the PThread is almost twice better. The reader should also be warned that small input sizes where the output is always produced in fractions of a millisecond are not a good

Table 2: PThread results with non-rectangular input size

	2000x100				*100x2000*			
Threads	***Time(S)***	***GFLOPS***	***Speedup***	***PTh/OMP***	***Time(s)***	***GFLOPS***	***Speedup***	***PTh/OMP***
seq	0.232	2.09	-	-	0.227	2.13	-	-
1	0.254	1.90	0.915	-	0.251	1.93	0.915	-
2	0.137	3.54	1.704	0.864	0.134	3.60	1.712	0.828
4	0.070	6.93	3.330	0.874	0.072	6.69	3.180	0.839
8	0.049	9.92	4.770	0.769	0.053	9.10	4.322	0.596
16	0.068	7.12	3.424	0.559	0.069	7.02	3.336	0.557
24	0.107	4.52	2.174	2.242	0.112	4.32	2.053	1.326
32	0.125	3.86	1.854	4.144	0.117	4.13	1.965	0.501

benchmarking scale for high performance application. The least execution time should always be at least a few seconds.

Furthermore, we can see that the other two sections, 5000 and 1000 sizes, are much better representatives and they commensurate accurately with our hypothesis in section 3.1. In both input sizes, the speedup of the two applications are almost the same (resulting in a ratio of almost 1). It can also be observed that while in 1000x1000 PThread is relatively dominant, the OMP implementation performed slightly better in 5000x5000. While the difference is not significant, it can be argued that for a larger task, the compiler has an easier job to optimize the loops because of the high length of the loop. In other words, since the OMP compiler is basically doing everything at its power to optimize the loops, this effect becomes more significant with larger loops. Hence, the OMP version is slightly faster in 5000x5000.

Similarly, we can see the results for the non-rectangular inputs in Table 2. As seen in the table, we observe a similar pattern to that of Table 1. The outputs of this table can be deemed as the intermediate step between 100x100 and 5000x5000 in Table 1. Since the input size is still not big enough, the gap between the PThread and OMP is slightly more noticeable, but still close enough to the general expected pattern explained in section 3.1.

Lastly, the same general conclusion obtained from the previous OMP survey, regarding number of cores, applies to PThread too. We see the best performance in almost all cases in executions with 8 threads. This is because of the CPU-intense nature of the tasks which makes only actual cores effective, limiting the speedup gain to practically 8 threads and no more. The same rule does **NOT** apply to tasks that are not compute-intensive. The histogram computation is an example of this and will be discussed in 3.4.

3.3 Results: PipeSort

Table 3 displays the measurement results of the PipeSort implementation. We conducted tests with up to 30000 elements and varying buffer size. It can be seen, that input sizes smaller than 1000 elements can be sorted within one second, irrespective of the buffer size. Beginning with 5000 elements execution times start to explode until a run with 30000 elements could not even finish

	Elements					
Buffer size	**100**	**1000**	**5000**	**10000**	**20000**	**30000**
1	0.01000	0.27821	7.25505	37.948	180.469	>900
5	0.00481	0.22224	3.90627	21.805	132.876	>900
10	0.00414	0.16936	2.73273	22.333	111.174	>900
50	0.00328	0.12219	2.83935	15.017	90.610	229.748
100	0.00262	0.09831	2.34266	14.002	74.173	>900
1000	-	0.03179	0.96372	5.483	42.331	120.881
10000	-	-	-	5.193	20.610	95.024

Table 3: Benchmark results of PipeSort. All values within the table are in seconds. > 900 means that the execution could not be finished within 900 seconds and was therefore terminated.

within 15 minutes on the DAS4 with buffer sizes 1, 5, 10, 100. Hence, the longest sequence of numbers our implementation manged to sort is 30000 with a buffer size of 50, 1000 and 10000.

It can be observed, that the bigger the buffer size the faster the execution. This behaviour is in line to our expectations. The bigger buffer size between the threads enables them to do more work until one thread starts to block because the buffer is full. This prevents too many expensive context switches and therefore speeds up the overall execution time. It would be interesting to further investigate how exactly the OS schedules the threads and if one can actually work until the buffer is full. Unfortunately, this exceeds the boundaries of this assignment.

The fact that runs with 30000 elements and a buffer size of 100 could not finish, but with a buffer size of 50 could finish, leaves us questioned. This behaviour is certainly not as expected and also does not support the general pattern the bigger the buffer, the faster the execution. A similar behaviour can be observed with 5000 elements with 10/50 buffer size and with 10000 elements with 5/10 buffer size. Here the difference is so marginal, that we can assume arbitrary measurement inaccuracies.

3.4 Results: Histogram

Table 4 demonstrates the results of the histogram calculation with different synchronization primitives over a 10000x10000 image. The common sequential execution time, used for speedup calculation, is `0.22 sec`.

As expected before, none of the approaches produce a real speedup. This is reasonable due to the heavy cost of blocking over synchronization barriers. To give an intuition about how costly this effect is, the image of this benchmark is composed of $100,000,000$ pixels. Dividing this number by the pallet size of the problem, 256, we obtain $390,625$. This number tells us that each cell of the pallet buffer, the histogram, is accessed on average more than 390,000 times. When accesses are split among multiple threads, many of them can become blocking, hence reducing the speedup significantly. Furthermore, we can observe that the atomic operation outperforms all other

Table 4: Benchmark of synchronized histogram. Sequential time 0.22 sec

	Atomic		*Mutex*		*semaphore*		*STM*	
Threads	***Time(S)***	***Speedup***	***Time(S)***	***Speedup***	***Time(s)***	***Speedup***	***Time(S)***	***Speedup***
1	0.791	0.238	2.029	0.093	2.116	0.089	8.749	0.021
2	2.039	0.092	2.276	0.083	2.119	0.089	41.003	0.005
4	1.243	0.151	3.418	0.055	4.148	0.045	28.211	0.007
8	1.051	0.179	3.589	0.052	3.614	0.052	21.553	0.009
16	0.943	0.199	2.918	0.064	2.818	0.067	14.665	0.013
24	0.917	0.205	2.701	0.070	2.853	0.066	17.119	0.011
32	0.946	0.199	2.645	0.071	2.812	0.067	33.019	0.006

primitives by a small margin. This can be argued by saying that with atomic operations, there is no *generic block scope*. In other words, the only critical section of the application is incrementing one cell of the histogram by one. The atomic operation implementation is the only implementation that protects this *and nothing more*. Hence, we can argue that it is reasonably the best performer. Furthermore, we can see that between mutexes and semaphores, mutex is slightly faster. This can be linked to the fact that a mutex is a *simpler form* of a semaphore, hence more efficient. Lastly, albeit having an optimistic approach that theoretically can lead to an efficient approach, we can see that the Software Transactional Memory approach is performing the worse among all. This can be linked to an unknown, non-efficient implementation of this concept in gcc and lots of collisions which need to be rolled back.

Nonetheless, in order to prove that this problem can also be parallelized efficiently we also provide the benchmarks of two parallel versions using the two approaches explained in 2.3, namely *image-value-parallel* and *image-size-parallel*. As explained, the former can work autonomously while the latter needs to store the histograms in a local buffer per thread and reduce them all at the end. These results can be seen in Table 5. As expected, the value-parallel implementation lies in the category of downgrade in terms of speedup, similar to implementations with synchronization. On the other hand, the native parallel implementation, size-parallel, shows the performance of a reasonably scalable parallel implementation. Furthermore, unlike the heat problem, since threads are spending most of their time reading and writing from the memory, we can see that the speedup does continue to improve on more than 8 threads, which is the actual number of cores. The fact that there is still improvement up to 32 threads, even though there are only 16 hardware threads available, is most likely due to clever scheduling and overlapping of the threads when accessing memory.

Aside from the observations that we have obtained from Table 4, we should also mention that these results are obtained using the most optimized implementation. A clear example of this is the mutex version. A rather naive approach toward implementing a lock with mutex is to create one global mutex for the entire histogram. This is inefficient, because there is no reason to block two threads that are writing to different indexes of the histogram. Instead, what we have done is to create an array of mutexes, each corresponding with one of the indexes of the histogram. Each thread will only try to obtain the lock associated with that particular index. In Table 6, we

Table 5: Benchmark of NOT synchronized histogram. Sequential time 0.22 sec

	Size-Par		*Value-Par*	
Threads	***Time(S)***	***Speedup***	***Time(S)***	***Speedup***
1	0.135	1.635	0.188	1.171
2	0.097	2.270	1.301	0.169
4	0.074	2.964	1.182	0.186
8	0.073	3.009	1.041	0.211
16	0.045	4.911	1.003	0.219
24	0.040	5.560	1.329	0.166
32	0.039	5.663	1.503	0.146

Table 6: Comparison of optimized and normal version of mutex

	Global Mutex		*Mutex Array*	
Threads	***Time(S)***	***Speedup***	***Time(S)***	***Speedup***
1	2.416	0.078	2.029	0.093
2	10.881	0.017	2.276	0.083
4	27.782	0.007	3.418	0.055
8	19.805	0.009	3.589	0.052
16	20.173	0.008	2.918	0.064
24	19.823	0.009	2.701	0.070
32	23.411	0.009	2.645	0.071

compare the benchmarks of these two implementation.

As clearly seen in the table, the performance gap is significant, almost in the order of being 10 times faster. Nonetheless, we can see that the relative speedup of both implementations with respect to the sequential implementation is low. Lastly, we have also conducted experiments for other sizes such as 50000x50000 that are not included in the report for the sake of consistency. The conclusions obtained from these input sizes were the same as the ones mentioned in the report and we have found the 20000x20000 size to be a better representative. The reader can refer to [4] to see the full results.

4 Conclusion

We have seen that PThreads is a more general, comprehensive and flexible alternative to OMP for multi-threaded programming. While being a more burden in terms of development, we have also seen that it enables the programmer to take full control over the behavior of the application. As an example, creating a big parallel region in OMP can be quite difficult to achieve while with PThreads, it is completely up to the programmer. Furthermore, we have seen that this flexibility enables us to use threads for a wider scope of applications. As an example, PipeSort could not have been easily implemented with OMP, while with PThreads it is completely feasible to do so.

On the other hand, we have seen that OMP does a good job at creating optimized parallel codes for well-known program pattern such as `for loop`s. This is why in most cases the PThreads implementation of heat dissipation is almost as good as OMP, but barely better than it.

Some other conclusions obtained from the tasks of this report can be enumerated as:

- Among the synchronization primitives, due to their simplicity, atomic operations are the best among all others. Mutex and semaphores are strong primitives but they are more general purpose than just performing an atomic operation. The software transactional memory, despite having a promising definition, does not seem to have a good implementation in gcc.
- As already seen in the second assignment and again in the PipeSort experiment, it is not a good idea to create too many threads. Context switches are very expensive and if there are lots of threads the OS is already very busy with only managing these without actual work being done.

References

[1] OpenMP http://www.openmp.org/

[2] https://gcc.gnu.org/wiki/TransactionalMemory

[3] https://gcc.gnu.org/onlinedocs/gcc-4.1.0/gcc/Atomic-Builtins.html

[4] Full experiment table, Google Docs