University of Amsterdam

VU Vrije Universiteit Amsterdam

OpenMP Parallelization

Exploring heat dissipation and sort problem

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 OpenMP 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. One of the most famous industry solutions to multi-core programming is OpenMP. OpenMP is a pragma based C/C++/FORTRAN library that enables simple and easy parallelization of applications. Unlike explicit parallel programming models (such as *pthreads* and *mpi*), OpenMP has implicit approach toward parallelization where the programmer hints the compiler, using `#pragmas`, about where and how to parallelize the code. The rest is handled by the compiler and the generated executable code can run on multiple CPU cores [1].

While most OpenMP introductions, because of focusing on iterative tasks, give the imagination that *loop parallelization* is the only type of supported code, this claim is not correct. OpenMP can support various constructs that enable all different kinds of parallelism. Three main types of parallelism that are explored in the assignment are: *Data Parallelism, Task Parallelism, Nested Parallelism.*

To Demonstrate the abilities, shortcomings and efficiency of OpenMP, three tasks are implemented using this library:

- **Heat Dissipation**: An iterative matrix cell update process based on neighbouring cells. This program only contains data parallel pragmas.
- **MergeSort**: A Divide and conquer sorting algorithm, implemented in both sequential and task parallel model.
- **VectorSort**: Sorting a vector of vectors to demonstrate the combination of data and task parallel model.

For the rest of this report, in section 2 we will explain the implementation details of each of these tasks, with respect to their programming model (*e.g* task parallel). In section 3 we will evaluate the performance of each implementation and compute their speedup based on a specific baseline. Furthermore, in section 3.4.2 we will analyze the performance of the heat problem symbolically and compare it with our empirical results. Finally, in section 4 we conclude about OpenMP and its applications.

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 MergeSort

The sequential implementation of the merge sort algorithm is integrated in the assignment's provided framework. It is mainly based on the top-down implementation presented on Wikipedia [2] with some optimizations (reduction in strength by bit shifting, code motion in loops) regarding performance. In principle the implementation does the following:

- Recursively divide unsorted list into sublists until a list only contains one element.
- Repeatedly merge sublists and create sorted sublists until only one sorted list remains

At the end of an execution the program reports the input size, the time needed to sort and sorted elements per second. Furthermore, the sorted list is automatically checked for correctness.

The parallel version is implemented using *OpenMP* task parallel constructs. By using this approach the code of the sequential and the parallel version is almost the same, except the pragmas for OpenMP and a threshold condition from where code gets executed sequentially to avoid too much parallelism.

The following code creates a parallel region with a given number of `threads` and shared pointers to the unsorted list `v` and a copy of it `b`. The parameter `l` describes the length of the list. Then, the `split_parallel` function is called by a single thread.

```
    [...]
    #pragma omp parallel shared(v, l, b) num_threads(threads)
    {
    #pragma omp single
        {
            split_parallel(b, 0, l, v);
        };
    }
    [...]
```

The subsequent code extract shows the `split_parallel` function, which recursively divides sublists. It can be observed, that the parallel recursion is stopped and execution continues sequentially when there are less than `1000` elements in a list. This threshold was chosen after extensive experimentation which is described in detail in section 3.2. Next, there are two tasks created with a recursive call to `split_parallel`, followed by a taskwait pragma, which assures that execution will only continue if both tasks have reached it. In the end the two sublist are merged together.

```c
void split_parallel(int *b, long low, long high, int *a) {
    // parallelism threshold
    if(high - low < 1000) {
        split_seq(b, low, high, a);
        return;
    }

    // recursively split
    long mid = (low + high) >> 1; // divide by 2
#pragma omp task shared(a, b) firstprivate(low, mid)
    split_parallel(a, low, mid, b); // sort left part
#pragma omp task shared(a, b) firstprivate(mid, high)
    split_parallel(a, mid, high, b); // sort right part

#pragma omp taskwait

    // merge from b to a
    merge(b, low, mid, high, a);
}
```

2.2 VectorSort

The vector sort implementation consists of sorting not just one, but sorting an array of vectors in parallel. The goal is to use both task and data parallelism. Based on this guideline, the overall plan of the implementation becomes trivial: the top-level vector is given to threads to parallelize the data. Consequently, each thread will receive a portion of the main vectors to sort and will sort each of them using a task parallel model. The following snippet demonstrates the overall structure of the above explanation.

```
#pragma omp parallel num_threads(DATA_THREADS)
{
   #pragma omp for
    for (long row = 0; row < rows; row++) {
      #pragma omp parallel num_threads(TASK_THREADS)
      {
         #pragma omp single
            {
               split_parallel(b[row], 0, length, vector[row]);
         };
      }
   }
}
```

Note that in this snippet, `b` is a pointer to an array of vectors, not a vector. Despite being self-explanatory, there is a small, yet very important detail in this snippet. All created *thread teams* have a fixed size using the `num_thread()` clause. Furthermore, the snippet is creating a nested parallel region that only works if `omp_set_nested(1);` is called. An example of the execution of the above snippet with `DATA_THREAD = 2, TASK_THREAD = 4` could be explained as follows: The master thread will create two parallel threads that reach the `omp parallel for` region and split the matrix rows into two sub-spaces. Then, each thread will go on and spawn four new threads as new workers. Consequently, one of these newly created threads will initiate the parallel job by calling `split_parallel();` and the other three will stay to receive tasks in the recursive execution of `split_parallel();`.

2.3 Heat Dissipation

The implementation of the heat dissipation problem with openMP is quite straightforward. The program consists of one main loop that has the potential of being parallelized by OpenMP. For this implementation we have used a modified version of the reference implementation provided by the framework and applied minor tweaks to it to reach the best performance. The main change to the reference code is eliminating the expensive *modulo* operation by first checking the `p->printreport` flag. If the option is not enabled, the modulo operation will never be executed.

Enabling parallelization in the heat problem is quite trivial. The following snippet will demonstrate how this goal can be achieved.

```
for (int iter = 0; iter < maxiter; iter++) {
#pragma omp parallel for \
        private(i, j)\
        schedule(guided)\
        reduction(max: maxdiff)\
        num_threads(p->nthreads)
    for (i = 1; i < h - 1; ++i) {
      for (j = 1; j < w - 1; ++j) {
        // update value
        // diff calculation
      }
    }

    // global diff check for termination
}
```

What this pragma does is essentially splitting the data into multiple sub matrices, each being updated by one of the threads. Furthermore, since the value of each cell could be dependent on its neighboring cells, a synchronization barrier is needed. Thankfully, the `#omp for` pragma contains an implicit barrier that synchronizes all threads at the end of each iteration.

The only aspect that need more attention is the `maxdiff` calculation. This variable cannot be shared among all threads because all of them need to both read and write from it. Furthermore, it cannot be private because at the end of each iteration the global `maxdiff` value needs to be calculated and there is no communication channel between a thread and the sequential region after it. The OpenMP `reduction(max: ...)` operation is built exactly for this scenario. The clause will basically make the variable private to all threads and the one with the maximum value will be store in a copy of the variable in the shared sequential region.

Lastly, the code snippet shows `guided` to be the chosen scheduling approach. The reader should be aware that this policy is an example in the code and has not been used in all tests. While in most cases this scheduling policy performed better than the rest, we have compared it to other policies to further justify this. Section 3.4 will go into the details of this evaluation.

3 Evaluation

This section first describes the experimental setup and the procedure of taking measurements. 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 DAS 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**: Best performance with 8 threads, each running on a single core without hardware threading because it is a floating point intense application. We expect a slow-down with 16 threads because of the scheduling overhead and the fact that effectively only one thread can be executed.
- **MergeSort**: Best performance with 16 threads, 2 each running on a single core with hardware threading because it is non floating point intense and therefore we expect the hardware threads to be useful.

- **VectorSort**: Best performance with 8/16 data parallel threads and 2 task parallel threads. So a team of 2 threads can run on a single core with hardware threading and share the closest cache. Again, because this is non floating point intense, we expect the hardware threads to be useful.

3.2 Results: MergeSort

Tables 1, 2 and 3 display the results of extensive testing of different parallelism thresholds on different input sizes. The speedup values are calculated in regard to a sequential execution on the respective input size. The parallelism threshold prevents the creation of too many small tasks, but on the other hand a to coarse threshold could lead to a slower execution because of sequential execution.

Multiple observations can be derived from table tables 1, 2 and 3:

- For a small problem size like 10^4 elements the overhead of creating and maintaining threads gets bigger with the amount of threads, thus resulting in less speedup. Also it needs to be noted that for a parallelism threshold $T = 100000$ the execution technically is sequential and for $T = 10000$ only 2 tasks are created, which can also be observed in the corresponding columns of table 1.
- Executing the parallel implementation with one thread does not come with too much overhead since speedups are around 1. An exception is the execution with $T = 100$ where the speedup is slightly lower. This is due to the fact, that with this small threshold a lot of tasks need to be created, which results in more scheduling overhead.
- In general it can be observed, that a threshold of $T = 1000$ and $T = 10000$ result in the best execution times. Only in a few cases execution with $T = 1000$ provides significantly better performance.

Based on these observations we decided that a parallelism threshold $T = 1000$ provides a good trade off between overhead for creating and executing tasks and task granularity on the DAS4. For another system, it might make sense to adjust this threshold depending on the compute power of a single (hardware) thread. A bigger task size can make sense if there is more compute power available per thread, which results in faster sequential execution time, thus creating too many tasks would have more overhead.

Figures 1, 2 and 3 display speedup graphs of executions with different input sizes 10^4, 10^6 and 10^8 in ascending, descending and random order, respectively. A few general patterns can be observed:

- Runs with a small input size of 10^4 independently from the initial order are performing good with 2 or 4 threads. When more than 4 threads are used the performance drops significantly.

Table 1: Comparison of parallelism threshold T on a initially randomly sorted list with 10^4 elements. Sequential execution time(s): 0.001010

	T=100		*T=1000*		*T=10000*		*T=100000*	
Threads	**Time(s)**	**Speedup**	**Time(s)**	**Speedup**	**Time(s)**	**Speedup**	**Time(s)**	**Speedup**
1	0.001063	0.95	0.001006	1.00	0.000999	1.01	0.001024	0.99
2	0.000716	1.41	0.000657	1.54	0.000635	1.59	0.001059	0.95
4	0.000824	1.23	0.000624	1.62	0.000694	1.46	0.001108	0.91
8	0.001307	0.77	0.000743	1.35	0.000746	1.35	0.001165	0.87

Table 2: Comparison of parallelism threshold T on a initially randomly sorted list with 10^6 elements. Sequential execution time(s): 0.14494

	T=100		*T=1000*		*T=10000*		*T=100000*	
Threads	**Time(s)**	**Speedup**	**Time(s)**	**Speedup**	**Time(s)**	**Speedup**	**Time(s)**	**Speedup**
1	0.15028	0.96	0.14429	1.00	0.14593	0.99	0.14797	0.98
2	0.08675	1.67	0.07818	1.85	0.07804	1.86	0.07765	1.87
4	0.06921	2.09	0.06163	2.35	0.06236	2.32	0.06536	2.22
8	0.05588	2.59	0.04011	3.61	0.04464	3.25	0.04498	3.22

Table 3: Comparison of parallelism threshold T on a initially randomly sorted list with 10^8 elements. Sequential execution time(s): 18.9637

	T=100		*T=1000*		*T=10000*		*T=100000*	
Threads	**Time(s)**	**Speedup**	**Time(s)**	**Speedup**	**Time(s)**	**Speedup**	**Time(s)**	**Speedup**
1	19.3134	0.98	18.9312	1.00	18.8926	1.00	18.9812	1.00
2	10.3519	1.83	10.0945	1.88	10.0106	1.89	10.1276	1.87
4	8.0176	2.37	7.6703	2.47	7.6830	2.47	8.2035	2.31
8	5.0843	3.73	4.3054	4.40	4.9649	3.82	5.1594	3.68

This is due to the overhead of managing and scheduling more threads on a rather small problem.

- Runs with a bigger input size yield the highest speedups. This makes sense because the time spent for scheduling and maintaining threads and tasks gets smaller compared to the time needed to solve the problem.
- For runs with 10^6 and 10^8 the speedup increases up to 32 threads and slowly declines for higher thread counts. The highest measured speedup was 6.9 with 32 threads and a random initial order.
- Runs with random initial ordering yield slightly higher speedups than descending and ascending order, which is caused by the count of necessary merging operations on the data which is highest with the random initial ordering.

Most interesting is the fact that the speedup increases up to 32 threads and only starts declining afterwards. Because of the 16 available hardware threads on the DAS4, we expected the highest speedups with 16 threads and a declining performance with more than 16 threads. We assume that the scheduler does adroit optimizations like overlapping task executions which leads to better speedups. However, a further investigation how exactly this behaviour can be explained would exceed this assignment.

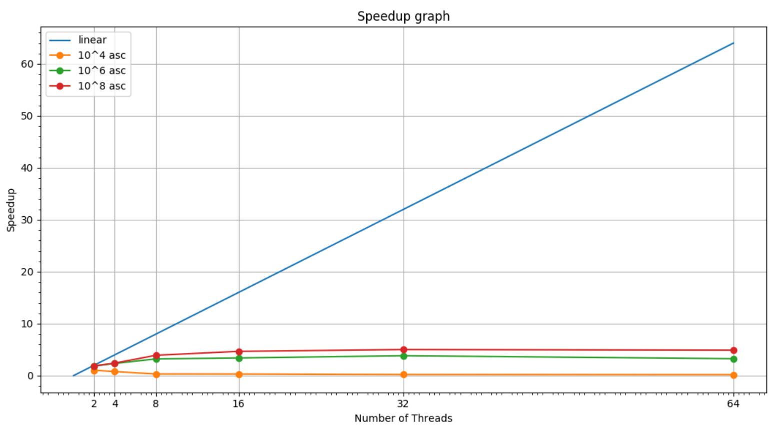

Figure 1: Speedup graph of different input sizes initially sorted ascending.

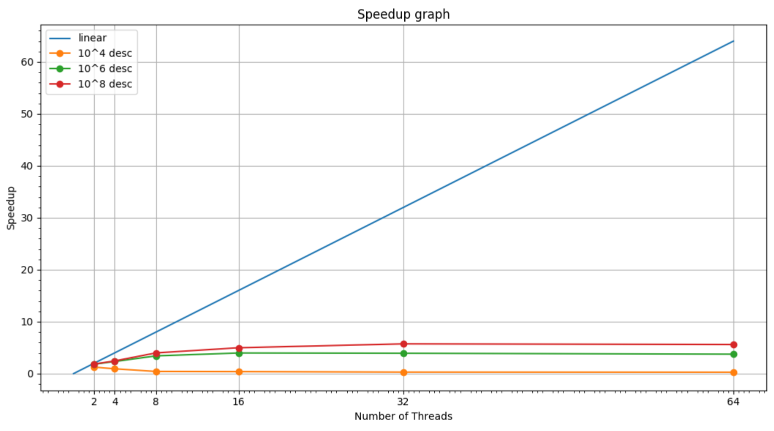

Figure 2: Speedup graph of different input sizes initially sorted descending.

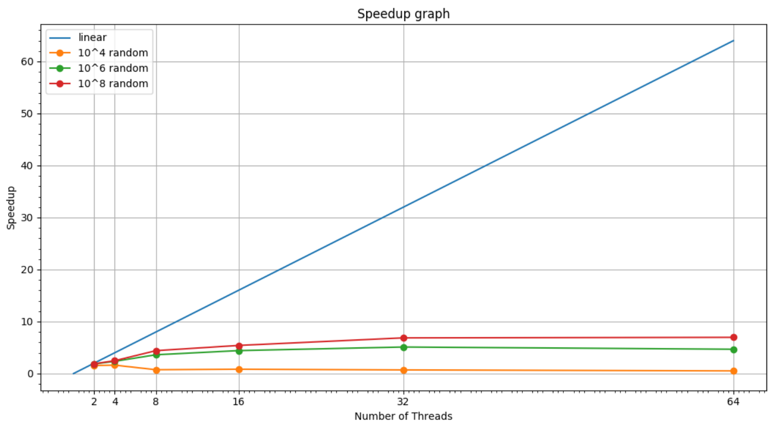

Figure 3: Speedup graph of different input sizes initially sorted randomly.

3.3 Results: VectorSort

Table 4 summarizes the most important results of the vector sort implementation. Before diving into the detail, the following notes should become clear:

- The *Fixed Size* version creates an array of fixed size vectors. In all examples, 100 vectors of 10000 elements is created, summing up to 1000000. For this scenario, we can argue that since the overall computation of each sub-vector is the same, static is the efficient scheduling and only these results are included in the table.

Table 4: Overview of Vecsort results

	Fixed Size- Static Scheduling		*Dynamic Size - Static Scheduling*		*Dynamic Size - Guided Scheduling*	
Threads	***Time(S)***	***Speedup***	***GFLOPS***	***SPEEDUP***	***GFLOPS***	***SPEEDUP***
0/0 (seq)	15.135	-	13.971	-	13.971	-
2/0	7.658	1.98	8.035	1.74	6.881	2.03
4/0	3.828	3.95	3.828	3.65	2.899	4.82
8/0	1.997	7.58	1.488	9.39	1.478	9.45
16/0	1.344	11.26	1.503	9.30	0.993	14.07
2/2	4.034	3.75	4.099	3.41	3.026	4.62
2/4	3.256	4.65	3.018	4.63	2.491	5.61
2/8	2.299	6.58	1.782	7.84	1.845	7.57
4/2	2.023	7.48	1.553	9.00	1.560	8.96
4/4	1.897	7.98	1.554	8.99	1.426	9.80
4/8	1.434	10.56	1.082	11.01	1.094	12.76
8/2	1.375	11.01	1.025	12.91	1.020	13.69
8/4	1.336	11.33	0.990	11.33	1.002	13.95
8/8	1.309	11.57	0.985	11.57	0.991	14.10

- The tests in *Dynamic* section sort an array of vectors of variable length with the upper bound of 1000000 elements. The range of numbers can be from 500000 to 1000000. All tests have been designed in a way that the the total number of elements created is the same, namely 75998877. Nonetheless, this concludes that the both columns of the Fixed size and Dynamic size section should *NOT* be compared. For these categories of tests, we have included both Static and Guided scheduling since the outcomes are actually different.
- The `X/Y threads` notation means X threads used for data parallelism and Y threads used for task parallelism. Furthermore, the `X/0` section refers to a version of the code that does only data parallelism.

Multiple facts can be concluded from table 4:

- For data parallel only versions, we see a different pattern than the heat problem and the upper bound of threads is not 8 (see section 3.4), rather it is more similar to that of the merge sort. Since the operation of sorting is actually not a matter of computation but rather comparison, not all threads need to have a computing unit. This makes simulated threads also effective. Hence, we can observe an **increasing** trend in the speedup up until 16 threads.
- A few superlinear speedups can also be linked to the previous fact. In the data-parallel-only section, we observer that for 4 and 8 threads, we get a higher speedup than the number of cores. While not justifiable, empirical experience shows that this is expected for tasks that are not compute intensive. In other words, since computation is not the bottleneck, the operating system can use multiple other parallelization mechanism (such as pipelining) to execute instructions in parallel.
- In the hybrid approaches, we see a somehow similar trend. Furthermore, we can see that in small data parallelisms, task parallelism can be effective. Albeit the scale of its effect is

less than that of data parallelism. On the other hand, we observe that with a high degree of data parallelism, the problem reaches its bottleneck and task parallelism becomes less and less effective.

- As expected, we can see that for arrays of dynamic size vector, the guided scheduling works better in almost all cases.

3.4 Results: Heat Dissipation

In this section we will describe the benchmarking results of the parallelized heat problem with OpenMP. First, we will display the obtained results of this application with different input sizes and variables. Next we will compare and evaluate these results with two performance analysis approaches, namely Amdahl's law and Operation Complexity.

3.4.1 Benchmarks

All benchmarks for the heat problem have been executed with the following parameter set:

- *200* iterations.
- *0.0001* convergence threshold to ensure maximum iterations being reached.

Furthermore, all speedups are measured based on the sequential version (`0(seq)` row). This is to keep the tables concise and readable as possible. The results show that sequential version and 1 threaded version perform almost identical, hence choosing either as a baseline is acceptable [1].

The only other variable in benchmarks is the *schedule()* clause in the parallel for pragma. In all of the graphs and tables below the scheduling will be noted. Furthermore, a dedicated section will compare different thread scheduling policies. Moreover, in all tests, *Thread Count 0* indicates the *sequential* implementation and *Thread Count 1* indicates execution of the *parallel* implementation with only one thread.

First, we will start by enumerating the result of the parallelized version with various different sizes. Table 5 demonstrates the performance of this implementation on different sizes and different thread numbers. Table 6 shows the same benchmarks for matrices with non-rectangular shapes. All tests are performed with *guided* scheduling.

Multiple observations can be derived from tables 5 and 6.

- In almost all cases, the speedup in leaner up until 4 to 8 threads. This commensurates with the description given in section 3.1. Due to the fact that not all CPU cores support float units, in reality, at most 8 of them can work in parallel.

[1]The complete list of benchmarks can be seen in [4]

Table 5: Benchmark results on rectangular input matrices with *guided* scheduling

	100x100			*1000x1000*			*5000x5000*		
Threads	**Time(s)**	**GFLOPS**	**Speedup**	**Time(s)**	**GFLOPS**	**Speedup**	**Time(s)**	**GFLOPS**	**Speedup**
0 (seq)	0.011604	2	-	1.195552	2.03	-	30.28982	2	-
1	0.01173	2.051	-	1.186489	2.049	-	29.54706	2.051	-
2	0.006277	3.77	1.86873	0.59407	4.093	1.99722	15.57455	3.9	1.89714
4	0.004414	5.43	2.65745	0.320411	7.59	3.70302	8.36254	7.28	3.53326
8	0.003573	6.23	3.28296	0.203026	11.97	5.84402	6.813173	8.92	4.33675
16	0.005478	4.23	2.14129	0.224021	10.085	5.29633	7.24	8.37	4.08109
24	0.092657	0.26	0.12660	0.321213	7.57	3.69378	7.46	8.14	3.96073
32	0.022183	1.09	0.52878	0.262459	9.26	4.52066	7.2	8.43	4.10376

Table 6: Benchmark results on non-rectangular input matrices with *guided* scheduling

	2000x100			*100x2000*		
Threads	**Time(s)**	**GFLOPS**	**Speedup**	**Time(s)**	**GFLOPS**	**Speedup**
0 (seq)	0.232694	2	-	0.227525	2.13	-
1	0.235	2.072	-	0.23	2.11	-
2	0.118	4.097	1.99153	0.11	4.23	2.09091
4	0.0611	7.95	3.84615	0.06	8.041	3.83333
8	0.0375	12.97	6.26667	0.0314	14.22	7.32484
16	0.038	12.57	6.18421	0.038	12.79	6.05263
24	0.24	2.01	0.97917	0.147	3.28	1.56463
32	0.52	9.28	0.45192	0.058	8.33	3.96552

- The Performance increase up until roughly CPU cores. In most cases, after this number of cores the overhead of managing parallelism becomes a bottleneck and prevents any speedup. In fact, in some cases this causes the performance to decrease. The best overall *GFLOPS* achieved is actually not achieved in these tables to preserve their homogeneity. With a matrix of size 2000 by 100 and using 16 threads, the application reached *16.8 GFLOPS*.

- Anomalies are also worth noticing. Our empirical experience with executing these benchmarks on DAS4 has shown that some of the execution times for the same parameter vary significantly. Albeit we have executed all tests 5 times and have taken the maximum, these anomalies still exist in our final benchmark. Moreover, since exhausting the number of cores of the machine (creating more threads than the number of cores) requires context-switching and thread scheduling by the operating system, these variations become more significant. Examples of these variations can be seen in the last rows of the table with thread numbers more than 16 *e.g.* thread number 24 of table 6.

- Focusing on the first two rows of each table, we can see that including the parallelization pragma in the code does not add a significant overhead. The measured values are either almost the same or slightly faster. This can be due to measurement inaccuracies.

One major dilemma that needs to be resolved and investigated is scheduling type. For this reason, we have executed some of the tests with *static* scheduling. Table 7 displays the FLOP

Table 7: Comparison of *static* and *guided* scheduling. All numbers are GFLOPS

	100x100		*1000x1000*		*5000x5000*	
Threads	**guided**	**static**	**guided**	**static**	**guided**	**static**
1	2.051	2.092	2.049	2.051	2.051	2.063
2	3.77	3.87	4.093	4.16	3.9	3.94
4	5.43	6.36	7.59	7.83	7.28	7.34
8	6.23	9.027	11.97	15.84	8.92	8.95
16	4.23	7.36	10.085	13.79	8.37	7.95
24	0.26	0.82	7.57	14.25	8.14	7.6
32	1.09	1.92	9.26	9.24	8.43	7.86

count of 100x100, 1000x1000 and 5000x5000 input matrices with static scheduling, comparing to guided scheduling.

Arguing about scheduling, based on Table 7, one might conclude that this specific task is a good suite for static scheduling since all iterations take approximately the same time. Furthermore, the numbers reported for static are also more intimidating. This argument holds only up to a certain point. Our observations, backed by the respective data tables, demonstrate the following facts:

- The static scheduling, on some arbitrary tests was able to reach much faster peaks.
- On the other hand, static scheduling had much sever variation in multiple identical executions.

Regarding the latter fact, we have stated so far in this report that *guided* is our primary choice of scheduling. This is only based on our preference and does not necessarily mean that this scheduling is always better. Nonetheless, depending on the requirements of the application and based on the mentioned facts above, one might opt to choose differently.

The reason for this behavior can be justified by noting two points. **First**, due to cache misses and memory bound nature of this task, variations are expected in the execution time. Furthermore, the *guided* scheduling, using a task dispatching mechanism, deals with this behavior better and leads to more stable results. The peak performances of the static scheduling are likely to be created in cases where memory access time of all iterations take approximately the same time, leading to all threads finishing at a synchronized time. **Second**, even the computation weight of iterations are not the same. A brief observation shows that in many generated pattern files, the conductivity of a cell is 0. This means that a sequence of addition and multiplications, will eventually be multiplied by zero. The compilers will automatically detect such patterns and replace their result with an immediate 0.

Lastly, the speedup graphs of static and guided scheduling is combined in Figure 4.

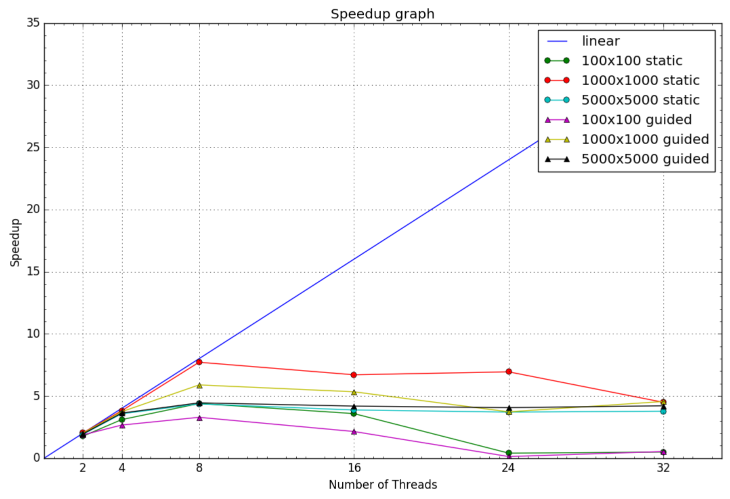

Figure 4: Speedup graph of static and guided scheduling

As seen in the figure, both scheduling types maintain a near-perfect speedup up until 8 nodes. Furthermore, we see a declining trend in all input sizes and scheduling policies. The only anomaly of this trend is 1000x1000 matrix in which the static scheduling performed better than the guided scheduling in most cases. Furthermore, this test sample was able to achieve a perfect linear speedup at 8 threads too.

Another interesting observation is that the negative overhead of parallelization is more significant in smaller matrices. This completely makes sense. In smaller programs, based on Amdahl's law, the parallelizable portion of the code is also smaller, hence the computation gain is also smaller and the overhead will become more significant.

3.4.2 Performance Analysis

Seeing the result of the benchmark, in this section we will compare those results against two famous performance analysis techniques, namely *Amdahl's law* and *Operation Intensity*.

Regarding **Amdahl's law**, we have modified the sequential version of the heat problem to calculate the time of the sequential and parallelizable portion of the program. The psudo-code for this measurement is as follows:

```
for (int iter = 0; iter < maxiter; iter++) {
   /* SEQUENTIAL BLOCK 1 */
    double maxdiff = 0.0;
    { void *tmp = src; src = dst; dst = tmp; }
    do_copy(h, w, src);
    /* SEQUENTIAL BLOCK 1 */

    /* PARALLEL BLOCK 1 */
    for (i = 1; i < h - 1; ++i) {
        for (j = 1; j < w - 1; ++j)
        {
         // Cell update
        }
    }
    /* PARALLEL BLOCK 1 */

    /* SEQUENTIAL/PARALLEL BLOCK 2 */
    for (i = 1; i < h - 1; ++i) {
        for (j = 1; j < w - 1; ++j)
        {
         // Diff update
        }
    }
    /* SEQUENTIAL/PARALLEL BLOCK 2 */

    /* SEQUENTIAL BLOCK 3 */
    if (maxdiff < threshold) { break; }
    if (iter % p->period == 0) { print_results(/*...*/); }
    /* SEQUENTIAL BLOCK 3 */
}
```

The execution time of each block is summed to a respective variable that indicates the execution time of sequential and parallel region. One significant flaw might caught the eye of the reader in this code. We have assumed the calculation of diff to be sequential. Nonetheless, in the real parallel code the calculation of the diff happens in the same parallel loop. This is because unlike cell updates which are independent in iterations, the diff update requires an expensive `reduction` operation at the end of the iteration to be executed [2]. Hence, assuming that the diff update is an **ideal parallel loop** is too optimistic. On the other hand, assuming that the diff update is sequential is also too pessimistic. To resolve this issue fairly, we have taken the time of this code region with a coefficient of `1/2` into account. Furthermore, we add the other `1/2` of this time to the parallel execution time. In other words, we assume that this region is parallelizable at best with a factor of half.

With these rubrics, we have measured the execution time of the same matrices. The outcome is as follows:

- **100x100**: `par 0.009986 / seq 0.001548 / sum 0.011534 / f 0.134212 / max speedup = 1/f = 7.450904`

[2]our experiments clearly showed that omitting them diff update in the loop increase its parallel speedup significantly

- **2000x100**: `par 0.218152 / seq 0.034190 / sum 0.252342 / f 0.135491 / max speedup = 1/f = 7.380579`
- **100x2000**: `par 0.216056 / seq 0.031677 / sum 0.247733 / f 0.127868 / max speedup = 1/f = 7.820595`
- **1000x1000**: `par 1.124710 / seq 0.193212 / sum 1.317922 / f 0.146604 / max speedup = 1/f = 6.821119`
- **5000x5000**: `par 28.385712 / seq 4.858965 / sum 33.244677 / f 0.146158 / max speedup = 1/f = 6.841926`

It can be seen that according to Amdahl's law, we can conclude that this problem is slightly better parallelizable in smaller input sizes. Furthermore, aside from arguing about the CPU type of the DAS4 machine that technically limits the maximum speedup, we can see that Amdahl's law is also suggesting that the speedup of this problem will not exceed approximately 8.

Comparing the expected speedup, we can see that in most cases the approximation is, at least in terms of upper bound, correct. As an example, all reported speedups of the two non-square matrix sizes with *guided* scheduling are within the approximated bound. The only exception is speedup of almost 10 for 1000x1000 input with *static* scheduling.

Regarding **Operation Complexity**, the first step is to count the number of operations in the parallel section of the code. Note that in this case, we assume that the diff calculation is a part of the parallel region. The outcome of the operation and memory access estimation is as follows:

- FLOPS: 1 negation for computing the rest of the weight. 8 sums and 3 multplications for cell update. `fabs()` and compare in diff. Total = 14.
- Memory Access: 8 for neighbor access. 1 for getting the previous value. 1 for getting the conductivity. 1 for storing the new value. Total = 10.

This yields the Compute intensity of **1.4**. Looking at the specification of the CPU used in DAS4 worker nodes [3], we also obtain the following information: the max memory bandwidth is `25.6 GB/s` and the peak floating point performance of the chip is `38.4GFLOPS`. Hence, using the following formula

```
maxspeedup = min(max_gflop, computeIntensity * max_memBandwidth)
```

we can obtain that maximum possible flop count of the machine is `35.84 GFLOPS` for this specific problem.

One might argue about the wide gap between this number and our obtained number. The best FLOP count obtained in our experiments is approximately 20GFLOPS. The difference can be explained by noting that we are not using all of the capacities of the chip in terms of parallelization.

Furthermore, similar to Amdahl's law, the reader should remember that the parallelization of the real code, unlike the assumption of this analysis, is **not ideal**.

4 Conclusion

In this assignment we used OpenMP `pragmas` to exploit data and task parallelism in three different C applications. Namely data parallelism in the heat dissipation, task parallelism in the MergeSort problem and both data and task parallelism combined in the VectorSort problem. Furthermore, we extensively tested these application regarding performance gains with different thread counts and possible bottlenecks. The most important findings are:

- For data parallelism, the number of available hardware threads (or cores) is a significant factor. As we have seen in the results, the speedup is bounded by the maximum number of physical threads that actually have a computational unit dedicated to them. Otherwise, although the program uses a higher level of parallelism, the thread will still get blocked for using the computational units of the CPU.
- The previous fact only holds for compute intensive tasks. The speedups of sorting problems, that are not computing anything and just comparing, demonstrates this fact. In other words, such tasks can use threads that do not have associated computational unit and benefit from a higher level of parallelism.
- Task parallelism is yet another useful approach for high performance computing. Albeit we have seen that its level of effectiveness is more limited when used alongside data parallelism.

Parallelization of existing sequential code with OpenMP is often trivial and the results can be game-changing for performance critical applications. If the application is suited for parallelization, a single `pragma` can yield almost linear speedups, given the right scheduling method is used. Since OpenMP 3 task parallelism is possible, which can also provide the application with significant speedup with only a few lines of code changes.

References

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

[2] Merge sort, Wikipedia https://en.wikipedia.org/wiki/Merge_sort

[3] Intel Xeon Processor E5620, Intel https://ark.intel.com/products/47925/Intel-Xeon-Processor-E5620-12M-Cache-2_40-GHz-5_86-GTs-Intel-QPI

[4] Full experiment table, Google Docs