Lab Project Report[*]

Assignment 2

Christopher Esterhuyse, Mihai-Andrei Spadaru

1 Introduction

Shared-memory computer architectures are extremely prevalent in all devices but the smallest or cheapest computers. With this surplus of logical cores available, it is prudent to build programs that can exploit this potential for parallelism. Although the more modern *multi-threading* approach offers an (arguably) more ergonomic means of achieving this over multi-processing, multi-threaded code is still more difficult than sequential code to implement correctly and efficiently.

Although there are endless means of arranging concurrent instructions to do work, many of the possible solutions can be covered by a set of generalized design patterns. An opportunity manifested: Expose these patterns as compiler directives (AKA 'pragmas') and delegate the nitty-gritty details of thread management to a compiler. OpenMP (Open Multi-Processing) describes an API for shared-memory processing. Several modern compilers such as C and Fortran now offer support for pragmas that implement this API.

In this report, we explore the effect of using OpenMP pragmas to multi-thread sequential C programs, each of which is the focus of its own section in Sections 2–4. focus on one of these programs each, where hypothesis are posed, reasoned about, and tested experimentally[1]. Succinct plots demonstrate experimental results, and are complemented by detailed tables of measurements in the Appendix.

[*]This report was submitted for the lab assignment of the 2018 edition of the Universiteit van Amsterdam Programming Multi-Core and Many-Core Systems master course.

[1]All included measurements were from executions run solo on a DAS4 node with 16 logical cores: 2 sockets, 4 cores per socket, 2 threads per core.

2 Heat Dissipation on a Cylindrical Surface

Data-parallelism describes the technique of achieving speedup by performing the same *task* over a partition of the problem domain in parallel. The means by which the problem is divided depends on the nature of the problem.

The simulation of heat dissipation on the surface of a cylinder is a good example of a problem that can be data-parallelized. Adapting the sequential implementation first requires identifying iterative loops without *loop-dependencies*. Although each simulation step must be executed in the correct order to preserve correctness, the same is not true for the ordering of surface points *within* one step. In this way, each thread can simulate a part of the surface independently.

Cylinders are represented *in silico* as matrices. As most input problems map to matrices with both dimensions significantly larger than the available number of threads, our parallel implementation subdivides the *outermost* loop (stepping over the height dimension) of a single simulation step ie. each thread simulates some rows of the matrix. As the work in different rows is both regular and uniform, *static* work scheduling is sufficient for load-balancing, and comes with the perk of minimal overhead.

Employing the OpenMP 'for' pragma requires deciding which variables within the loop are 'shared' (copied over threads by reference) or some variant of 'private' (thread-local). For most variables, the choice is clear to preserve correctness; however, whether the loop-invariant counter values should be 'shared' or copied locally for each thread (with 'firstprivate') was not immediately clear. The two options were compared head-to-head in a series of tests, the results of which are shown in Fig 1. It

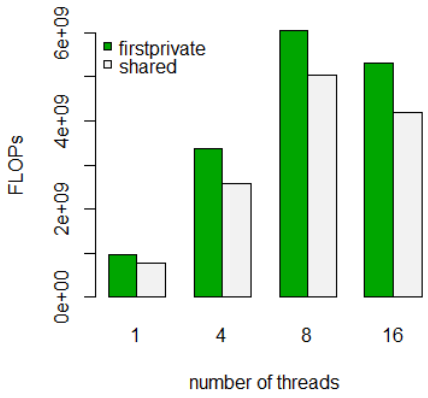

Figure 1: FLOPs achieved by two implementations using different numbers of threads. 'firstprivate' copies loop-invariant variables by value, and 'shared' copies loop-invariant variables by reference.

can be seen that the time taken to dereference these pointers was not negligible, giving the 'firstprivate' variation an edge. As such, our experiments onward used this implementation.

The spawning and joining of threads incurs overhead not present in the sequential implementation, begging the question: When is it worthwhile to multi-thread such a program? Impeded by thread-spawning overhead without any upside, we are confident the 1-thread OpenMP implementation will be outperformed by the sequential implementation. Exactly how many threads are needed for the former to overtake the latter isn't certain, but we expect that perhaps even 2 threads would be enough. In an attempt to answer this question, the parallelized program was run in all combinations of the following conditions:

1. Number of threads in $\{1, 2, 4, 6, 8, 32\}$
2. Input matrices with dimensions ('Input size') in $\{\ 100 \times 100,\quad 1000 \times 1000,\quad 100 \times 20000,\quad 20000 \times 100,\ 5000 \times 5000\ \}$

All input problems were generated from scaled[2] versions of the same initial temperature and conductivity images[3], seen in Fig 2. The results of the experiment can be seen in Fig 3.

(a) Initial temperature

(b) Conductivity

Figure 2: Initial temperature and conductivity inputs for all experiments in Section 2.

We observe that the sequential implementation is outperformed by the OpenMP version the moment it is provided at least two threads. No significant speedup was achieved using 16 threads over 8 threads. This can be attributed to double-precision floating points being a significant bottleneck in the execution, as the DAS4 node used only has the capability to truly support 8 such operations in parallel. The slowdown for 32 threads was also to be expected, as this exceeds the logical core count of the machine outright, resulting in threads sharing resources instead of running in parallel. For most of the input problems, however, the speedup from 4 threads to 8 was no longer linear. This outcome would be explained by other services running on the machine requiring some floating point operations, causing contention. We also believe that this non-linear speedup might be caused by the architecture of the DAS4's nodes. The 8 physical cores present on each node are distributed over two quad-core sockets. This might entail a slowdown due to communication overhead between the cores when running the process on more than 4 threads, since these two sockets do not share a fast cache memory.

3 Merge Sort

A program exhibits task-parallelism when it is capable of performing several heterogeneous tasks in parallel.

This section explores merge sort[4], a 'divide-and-conquer' sorting algorithm, in our case intended to

[2] All images were scaled using linearly-interpolated versions of a starting image with dimensions 500×500.

[3] These images were chosen as they exhibit both interesting edges and gradients, resulting in non-trivial simulations.

[4] Merge sort is described in detail on Wikipedia at `https://en.wikipedia.org/wiki/Merge_sort`

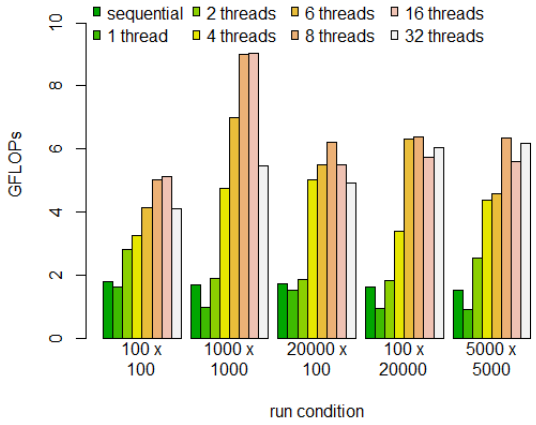

Figure 3: Performance of the temperature-simulation program (in GFLOPs). Each bar group represents a particular input problem size. Within each group, bars further right indicate execution with higher thread count. The leftmost bar is the sequential implementation for reference.

sort a vector of integers. The algorithm can be broken up into two sequential steps:

1. Split the input vector in half to produce two sub-vectors. Merge sort each sub-vector recursively unless the have the trivial length of 1.

2. Merge the two sorted sub-vectors into one sorted vector. Return the merged vector.

The first step described two independent, recursive merge sort sub-tasks, suitable for parallelization. Although there do exist non-recursive implementations[5] of merge sort, we adhere more closely to the classical definition for simplicity. Our implementation uses what is often called 'top-down' merge sort. It is not in-place, requiring a second 'working' buffer to mirror the input vector, allocated once at initialization. Our implementation comes with some interesting properties not common to all possible versions of merge sort:

1. At the beginning of the procedure, a **threshold value** for vector length is calculated. This value is compared to the current vector length during sorting to allow and suppress task-parallel thread spawning. In effect, this makes sure only the largest merge-splits generate threaded tasks to optimally balance work between threads.

2. During the merge step, the loop short-circuits to a **specialized 'fill' loop** once either sub-vector has been fully merged, filling[6] the remaining cells without further redundant conditional checks.

3. At each call depth, the two available **buffers are swapped**. Thus, the work of any sub-tasks two layers deeper is overwritten. For merge sort this isn't a problem, and it means the fresh data never needs to be copied back.

4. Very **short vectors** don't generate threaded sub-tasks when their length would result in a sub-problem that would likely be solved faster sequentially.

Either of the OpenMP pragmas 'taskgroup' or 'taskwait' could be used to achieve task-parallelism. The impact of this choice on runtime was measured with a small experiment, the results of which are shown in Fig 4. As anticipated, 'taskgroup' allowed for more fine-grained barriers for tasks, resulting in appreciable speedup over the 'taskwait' implementation. The final implementation uses this approach.

During the merge step of the merge sort algorithm, the current head elements of the two sub-vectors are repeatedly compared. If one sub-vector is fully merged into the new total vector before the other, no more comparisons are necessary. For this reason, the speed of a merge sort execution increases for each time one sub-vector finishes significantly before the other. This would be the case if the input vector were already sorted in either ascending or descending order. By contrast, the

[5]The algorithm is recursive by nature. Iterative implementations cannot avoid the resulting loop-carry dependencies, but they can potentially save on memory and time by not incurring costs associated with additional stack-frames.

[6]Two versions of this loop were tested: one using a for loop, and another batching the remaining writes using the library function memcpy. The former proved faster in every test we threw at it, and as such, was used for our implementation.

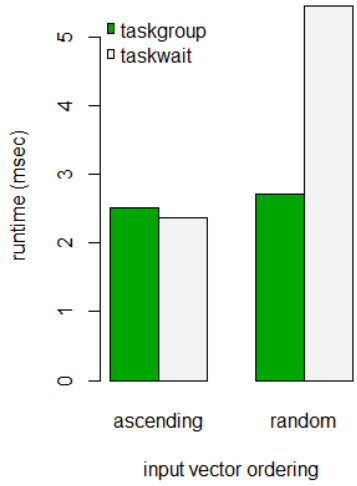

Figure 4: Measured runtime using two variations of a OpenMP-task-parallelized merge sort using 8 threads with an input vector of length 10^4.

worst case is expected to arise for vectors with orderings that result in both sub-vectors always finishing together (for example: $[8, 1, 7, 2, 6, 3, 5, 4]$[7]). Executions were timed for input vectors of lengths $\{10^n | n \in [4, 5, ..., 9]\}$[8]. To capture performance as a function of the initial ordering of vector elements, three representative input orderings were distinguished, shown (with short aliases for brevity) in Table 1.

Table 1: Labels and descriptions of the various representative input orderings for vectors used in the experiments of Section 3.

Label	Description
A	Elements sorted in ascending order.
D	Elements sorted in descending order.
R	Random integer elements.

The measured runtimes can be seen in Figures 5 – 7. As expected, the shortest input vectors did not benefit from excessive parallelization. Fig 5 shows how the threading-overhead balanced out with the benefits of parallelization at approximately 4 threads for vectors of length 10^4 (resulting in minimal runtime using 4 threads). For the longest vectors, shown in Fig 6, the opposite was true. Additional threads decreased the overall runtime. However, the benefits tapered off at 16 threads, as on the DAS4, threads would be distributed over 16 CPUs. With a different view on the same data, Fig 7 shows the difference in runtimes when specifically comparing input orderings. As before, increasing thread count beyond 16 made no appreciable difference[9]. As anticipated, the R input vectors resulted in the longest runtimes. However, the scaling factor from A to R was expected to be between $1\times$ and $2\times$. In several cases the perceived value was even higher. The reason for this is unclear to us. One idea that would be worth exploring is whether there is some degree of *dynamic conditional branch prediction* at work. The A and D inputs would be the best-case scenario for predicting the branch taken in the merge step, with left and right vectors always finishing first for those inputs respectively. By contrast, R vectors are unpredictable by definition, causing branch prediction to fail often.

While the difference is less striking, D vectors took slightly longer than A vectors. This might be explained by an implementation detail: writing from the right sub-vector (always occurs with D) requires an extra jump instruction than writing from the left sub-vector (always occurs with A).

4 Vector Merge Sort

In this section we explore a super-problem of that described in Section 3. This program takes as input a vector of integer vectors, and merge sorts each inner vector independently. The task-parallel nature of each nested merge sort problem is preserved. However, this homogeneous application of merge sort to several vectors at once affords leveraging data-parallelism as well.

For the sake of brevity in this section, we define

[7] An experiment to explore the time taken for vectors of this form was considered, but ultimately cut to save time and concentrate on exploring task-parallelism instead.

[8] Our attempt to include runs with inputs of length 10^{10} on DAS4 resulted in the process being killed by the administrator. Hence we deemed 10^9 to be sufficient.

[9] Runs using 32 and 64 threads should take longer in principle, but as the thread-spawning overhead for this run is identical to that for the runs in Fig 5 (where runtime is measured in *milliseconds*), the overhead is negligible at this scale.

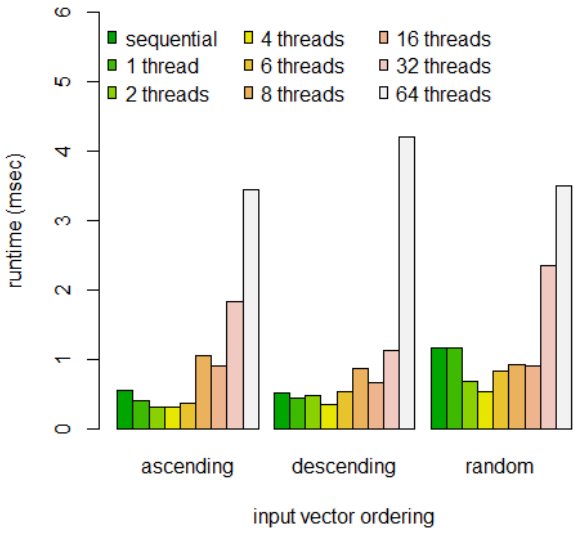

Figure 5: Runtime of merge sort with input vectors of length 10^4 (short). Groups distinguish input orderings. Within each group, the run conditions are laid out from sequential (left) to OpenMP with 64 threads (right).

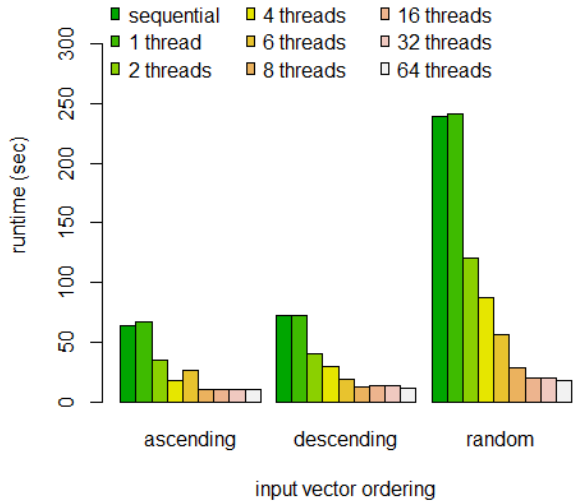

Figure 6: Runtime of merge sort with input vectors of length 10^9 (long). Groups distinguish input orderings. Within each group, the run conditions are laid out from sequential (left) to OpenMP with 64 threads (right).

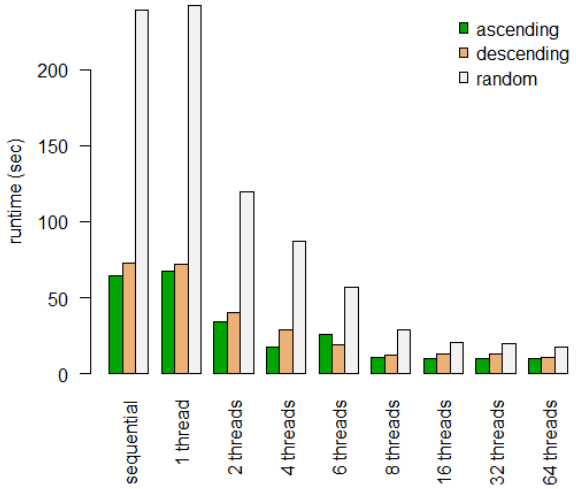

Figure 7: Runtime of merge sort with input vector of length 10^9. Bar groups distinguish between runtime condition (number of threads for the OpenMP implementation and the sequential implementation). Within each group, bars distinguish the initial ordering of the input.

the following notation: Vector-merge-sort problem instances are suggestively named 'm × avg. n', denoting a problem instance of m vectors, each of random length, sampled uniformly[10] from the range $[1, 2n]$.

In the previous sections, it was interesting to observe the changing behavior of the system when provided a different number of available threads. This provided insight into the tipping point of where the *extent* of the parallelism lead to an edge in performance over the sequential implementation, overhead and all. In this section, we take 8 threads[11] as granted and instead focus on exploring a different question: What is the best way to *use* our available threads? There exists a spectrum of possible implementations between pure data-parallelism and pure task-parallelism for this prob-

[10] The canonical use of something like 'rand() % n' to generate a random number on the interval [0,n) in C does not truly result in uniformly-distributed samples (without bias). However it is close enough for our purposes.

[11] Even though 16 threads would almost certainly run faster than 8 threads on DAS4, we used 8 to avoid other system services contending with resources (as observed from experiments in Section 3) while still providing considerable parallelism.

lem. To best represent this spectrum, our solution supports all three types of parallelism using different function implementations that can be set at runtime using a parameter. These implementations differ only in placement of OpenMP pragmas (and some necessary bookkeeping code to go with it). These implementations are enumerated along with short-hand aliases for brevity in Table 2.

Table 2: Labels and descriptions of the versions of the program used for experimentation in section 4.

Label	Description
D	Data-parallelism only.
T	Task-parallelism only.
H	Hybrid of data- and task-parallelism.

Once again (as in Section 2), we aim to incorporate data-parallelism into our OpenMP program. Before, the choice of 'static' scheduling was obvious, as work per iteration was uniform. Assuming this program will be used on inputs with *many* more random-length vectors per input than threads[12], one can rely on *the law of large numbers* for each thread to have approximately even work. For this reason we anticipate that 'static' scheduling is sufficient to decide how to partition the outer vector over threads. Our intuition was supported by the findings of an experiment, with results shown in Fig 8. It can be seen that there is no substantial gain to be had by using any of the other chosen scheduling schemes. Te opted to use static scheduling for the final implementation because of its simplicity.

We look first to Fig 9, showing a small set of 'balanced' problem inputs (where the *number* of vectors is comparable to the average *length* of vectors). It is clear that while D and H are comparable, T simply incurs too much overhead on thread-spawning. This is to be expected, as implementation T solves these problems with very many sequential encounters with merge sort, each incurring a barrier between all threads, with thread-tasks being created deep into the recursive tree for small sub-vectors.

[12]We deem this a reasonable assumption. The problem prompt makes no well-defined disparity between difference in orders of magnitude for the *number* of vectors and *lengths* of vectors. As only problems with significant *work* are interesting, this would suggest for most cases the number of vectors will overshadow the number of available cores for threads.

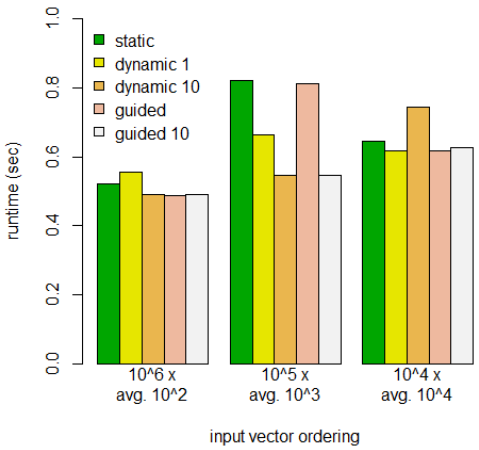

Figure 8: Runtimes of the data-parallel OpenMP version of the vector-merge-sort program. Each bar group represents an input problem Each input was initially sorted in descending order. Bars within groups discern between scheduling schemes.

We look next to Fig 10, where there are few vectors, each of which tends to be very long. Here T does best (particularly on the most extreme case). While there is still an order of magnitude difference between the other two problem instances in this figure, it suggests the *range* of input problems for which the trade-off between data- and task-parallelism balance out. It seems that the choice of how to spend one's threads here does not matter much.

Finally, in Fig 11 we observe measurements in response to an input problem with the most extreme bias toward *many* vectors of *small* average length used in our tests. Following the precedent set by the observations above, task parallelism is *by far* the poorest choice here, requiring a split y-axis to plot the results meaningfully.

Across all the experiments, we observe that H is never the optimal choice of the three. Most obvious from Figs 9 – 11 is that whether D or T is superior depends entirely on the 'shape' of the input problem. It follows then, that (in general) if the input problem defines a sort of *ratio* of effectiveness of data- and task-parallelism for each problem, that never would the optimal solution for a particular

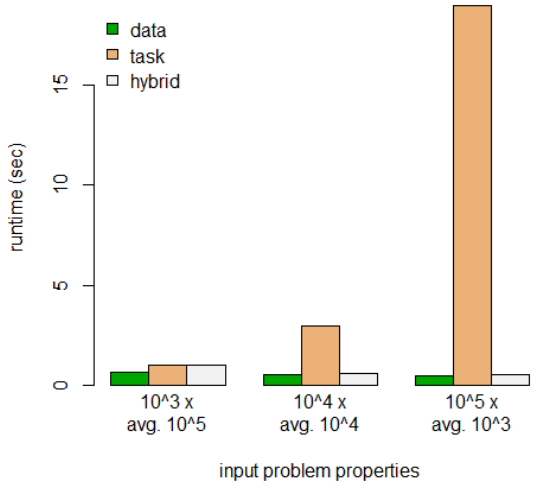

Figure 9: Measured runtime for three 'balanced' vector-mergesort problems (with comparable vector *average lengths* and *number*). Within each group, bars discern between the parallelization-approach used.

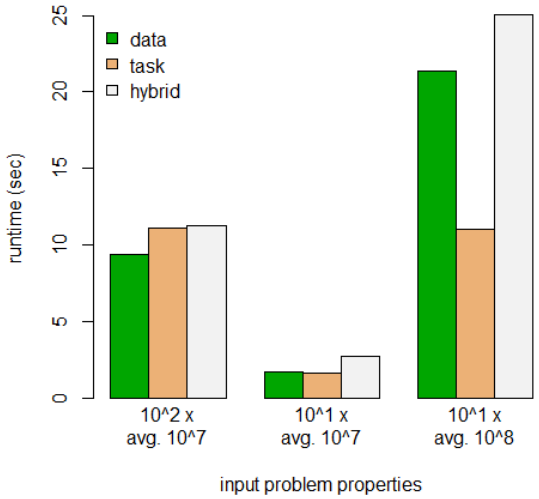

Figure 10: Measured runtime for three 'fewer vectors' vector-mergesort problems (with fewer, longer vectors). Within each group, bars discern between the parallelization-approach used.

problem be the hybrid approach. However, we also observe that while it was never the best, defaulting to hybrid-parallelization would also not result in the *worst* runtimes for the same reason, and might even be a suitable choice if one is not able to predict the properties of the inputs the program will typically be used to solve.

That said, data-parallelism seems to be superior for *most* inputs for our vector-merge-sort program in particular. We believe this is caused by the task-parallelization being nested *within* the data-parallelization (ie. the outermost loop is the data-parallelizable one). As such, this bias is the result of data-parallelization being synonymous with *coarser-grained* parallelization for this program. For other programs, where this order were inverted, we would expect the bias to also be inverted for this same reason.

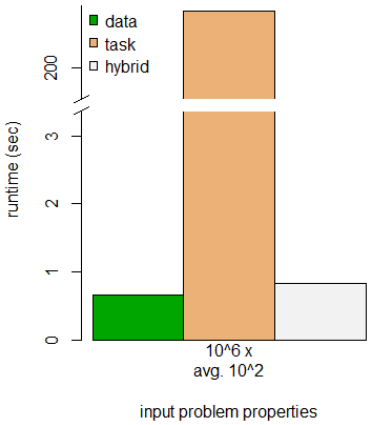

Figure 11: Measured runtime for one 'many vectors' vector-mergesort problem (with more, shorter vectors). Bars discern between the parallelization-approach used.

5 Conclusion

OpenMP is a powerful means of multi-threading a sequential program that exhibits meaningful concurrency. Its pithy pragmas carry a lot of functionality without introducing much development friction, allowing for rapid iteration on code for test-

ing the behavior of a program. The terseness and transparency of these pragmas are their greatest utility – but also their greatest danger, requiring the programmer to ensure that pragmas are neither failing silently, nor causing unintended behavior for either correctness or performance.

In this report we show our findings: Both data- and task-parallelized programs can be subject to considerable speedup without much added clutter to the sequential source code using OpenMP. For problems that facilitate either of these two approaches, we found that there is no option that is consistently superior for all possible problems. Rather, the nature of the program tends to lend itself toward one approach or another. For example: which parallelizable loop would result in the most coarse-grained thread-tasks?

Lastly, we demonstrated the objective superiority of the multi-threaded OpenMP programs over that of the sequential implementation when provided even a moderate number of logical cores. In all three cases, worthwhile speedup was achieved.

Appendix

Table 3: Precise values shown in 1 from Section 2. Columns show runs with the given thread-count. Each row shows GFLOPs achieved for an implementation variant. Each value shown is computed as the mean of 5 runs.

	1	4	8	16
firstprivate	0.961799	3.382641	6.046584	5.327694
shared	0.7705838	2.5899900	5.0324830	4.2007190

Table 4: Precise values shown in Fig 3 from Section 2. Each cell shows performance in GFLOPs for the mean of 5 runs. Columns separate thread counts, except for the first which shows the sequential performance for comparison. Each row represents an input problem with the indicated horizontal and vertical dimensions respectively.

	sequential	1 thread	2 threads	4 threads	6 threads	8 threads	16 threads	32 threads
100 x 100	1.798211	1.610953	2.813055	3.242791	4.142128	5.026718	5.118429	4.099109
1000 x 1000	1.675142	0.980945	1.903946	4.755560	6.992037	8.988076	9.032184	5.465888
20000 x 100	1.718381	1.502509	1.862262	5.024058	5.480072	6.215681	5.491176	4.912022
100 x 20000	1.623915	0.938974	1.823758	3.374927	6.307391	6.376013	5.735369	6.026893
5000 x 5000	1.519553	0.907138	2.530656	4.374141	4.581947	6.351311	5.603441	6.185713

Table 5: Runtime (in milliseconds) of values shown in 4, Section 3. Note that the rows with 1 thread were not used in the plot, as they were not particularly interesting.

input ordering	threads	taskgroup	taskwait
ascending	1	0.002405	0.002493
	8	0.002509	0.002363
random	1	0.009890	0.011602
	8	0.002722	0.005450

Table 6: Runtime (in seconds) from executions of merge sort in Section 3. Row groups distinguish between number of threads for the OpenMP implementation and the sequential implementation. Row groups contain one value for each ordering of the input vector (with $a = ascending, b = descending, r = random$). Columns denote the input vector length.

		10^4	10^5	10^6	10^7	10^8	10^9
sequential	a	0.000543	0.006142	0.047326	0.532971	5.825462	64.099548
	d	0.000508	0.004764	0.054654	0.633925	9.520850	72.68476
	r	0.001168	0.013715	0.243766	1.885519	21.205352	238.77072
1 thread	a	0.000407	0.004480	0.049503	0.582437	6.136688	67.342340
	d	0.000442	0.004950	0.053651	0.607699	6.558318	71.98732
	r	0.001159	0.014206	0.165446	1.918058	21.483068	241.77186
2 threads	a	0.000312	0.002419	0.025785	0.300166	3.160762	34.614656
	d	0.000478	0.003939	0.030136	0.352883	3.844725	40.32472
	r	0.000673	0.006899	0.121396	0.942474	15.828055	119.94618
4 threads	a	0.000307	0.001525	0.013473	0.155340	2.322190	17.989005
	d	0.000354	0.002452	0.015870	0.186413	1.972054	28.98639
	r	0.000522	0.003532	0.039799	0.461759	5.200550	87.42984
6 threads	a	0.000367	0.001391	0.012636	0.155991	1.642273	25.952306
	d	0.000527	0.001698	0.015301	0.180983	2.647837	19.07725
	r	0.000829	0.005196	0.038674	0.455079	5.234768	56.61428
8 threads	a	0.001045	0.001377	0.007685	0.090802	0.961804	10.719772
	d	0.000869	0.001946	0.008858	0.134629	1.141731	12.03896
	r	0.000924	0.002795	0.020108	0.237612	2.594323	28.96994
16 threads	a	0.000897	0.001208	0.028156	0.088120	0.985991	10.086905
	d	0.000660	0.001866	0.009934	0.152394	1.152961	13.31417
	r	0.000898	0.001881	0.022272	0.217298	2.819866	20.39448
32 threads	a	0.001838	0.001723	0.008555	0.086722	0.952655	10.411055
	d	0.001130	0.003446	0.012201	0.111365	1.061401	13.27704
	r	0.002357	0.004183	0.017719	0.171468	1.767524	20.26058
64 threads	a	0.003450	0.004504	0.011116	0.081467	1.016764	9.777334
	d	0.004199	0.004509	0.012654	0.095815	1.032882	10.79655
	r	0.003499	0.005623	0.019852	0.151697	1.773163	17.99900

Table 7: Runtime (in seconds) from executions of vector-merge sort in Section 4. Each row isolates a problem instance '$m \times$ avg. n' with m vectors, each of random length, sampled uniformly from the range $[1, 2n]$. Each column specifies an OpenMP scheduling scheme.

	static	dynamic 1	dynamic 10	guided	guided 10
$10^4 \times$ avg. 10^4	0.644923	0.617293	0.745140	0.615952	0.625059
$10^5 \times$ avg. 10^3	0.821476	0.664405	0.546425	0.812009	0.546555
$10^6 \times$ avg. 10^2	0.521070	0.554881	0.490840	0.487535	0.488908

Table 8: Runtimes (in sec) for the OpenMP implementation of vector-mergesort from Section 4. Results are given for columns D (data-parallel), T (task-parallel), and H (hybrid).

number of vectors	average vector length	D	T	H
10^1	10^7	1.768838	1.705830	2.781976
10^1	10^8	21.33670	11.02224	25.02424
10^2	10^7	9.411153	11.121160	11.279298
10^2	10^6	1.125674	1.402159	1.023721
10^3	10^5	0.686004	1.041928	1.054905
10^4	10^4	0.581549	2.955941	0.625004
10^5	10^3	0.499439	18.947617	0.581480
10^6	10^2	0.660314	201.220804	0.823140