Threaded Programming coursework I: benchmarking OpenMP schedules

Jonas Fassbender jonas@fassbender.dev

Abstract

Keywords:

1. Introduction

This paper documents the results of a benchmark performed on a scientific program. The program is written in the Fortran programming language and performs element-wise computations on matrices and vectors (using nested do-loops, not Fortran's array operations). It contains two of these matrix/vector operations—in this paper called critical sections.

Both critical sections are suitable for speeding up with OpenMP's loop construct, distributing the computation on multiple threads of execution. The loop construct provides the `schedule` clause, which determines the division of the loop-iterations among the OpenMP threads (see OpenMP Architecture Review Board, 2015, Chapter 2.7.1).

The benchmark consists of two phases. The goal of the first phase is to compare different scheduling options of the OpenMP library and how they effect the execution speed (measured in seconds) of the two critical sections of the program. The second phase provides data on how well the fastest scheduling options for both critical sections scale with different amounts of threads.

OpenMP version 4.5 was used and the benchmark was performed on the back end of the Cirrus supercomputer (see OpenMP Architecture Review Board, 2015; EPCC, 2019). The program was compiled with the Intel Fortran Compiler (`ifort`) version 17.0.2, with the maximum optimization provided (optimization level `O3`) (see Intel, 2016).

First, this paper describes the conducted benchmark, before presenting the results. At last the results are discussed and a conclusion is drawn.

2. Experiment

This chapter will describe the performed benchmark. First the two critical sections are described mathematically, followed by a description of the benchmark.

Let $n \in \mathbb{N}$ be a positive integer. Let $A : n \times n$ and $B : n \times n$ be two matrices, $A, B \in \mathbb{R}^n \times \mathbb{R}^n$. Let $A(i, j); 1 \leq i, j \leq n$ be the element of A in the ith row and the jth column. Every element in A is initialized to 0 and every element in B is set according to: $B(i,j) = \pi(i+j); i,j = 1,\dots,n$.

The first critical section updates A:

$$A(i,j) = A(i,j) + \cos(B(i,j)); i,j = 1,\dots,n. \tag{1}$$

Both critical sections are executed multiple times, which is the reason $A(i,j)$ on the right-hand side of (1) can not be substituted to 0.

For the second critical section, let $\vec{c}$ be the zero vector of size n. Let $\vec{j}_{\max} \in \mathbb{N}^n$ be another n-sized vector. $\vec{j}_{\max}$ is set to:

$$i = 1,\dots,n : \vec{j}_{\max}(i) = \begin{cases} n & \text{if } i \bmod 3\lfloor \frac{i}{30} \rfloor + 1 = 0 \\ 1 & \text{if } i \bmod 3\lfloor \frac{i}{30} \rfloor + 1 \neq 0 \end{cases}. \tag{2}$$

The matrix B' is set to $B'(i,j) = (ij+1)n^{-2}; i,j = 1,\dots,n$.

The second critical section updates $\vec{c}$:

$$\vec{c}(i) = \sum_{j=1}^{\vec{j}_{\max}(i)} \sum_{k=1}^{j} \vec{c}(i) + k\ln(B'(j,i))n^{-2}, i = 1,\dots,n. \tag{3}$$

Since both (1) and (3) are element-wise independent, the computation of every element can be distributed over multiple processes.

It should be noted here, that (1) is a perfect computational cube of $n \times n$. That means, that every iteration—each a manipulation of a single cell $A(i,j)$—contains the same amount of instructions, making it trivial to split the iterations and having a balanced distribution of work per thread.

On the other hand one can see that (3) does not behave in the same way. Each iteration is dependent on $\vec{j}_{\max}(i)$, which is not constant. Instead, $\vec{j}_{\max}(i)$ equals either 1 or n and the distribution of $\vec{j}_{\max}(i) = n$ is asymmetric, since the modulus in (2) changes depending on the iteration i. For example, $\vec{j}_{\max}(i) = n$ for every element in the interval $i \in (1, 29)$, while $\vec{j}_{\max}(i) = n$ is true for only 7 elements in $i \in (30, 59)$ and the amount keeps decreasing with bigger i. This has the consequence, that the first iterations are computationally more heavy and time consuming than the later iterations.

The benchmark consists of two phases. In the first phase, different scheduling options are compared using four threads and the fastest scheduling option for each critical section is determined. The second part of the benchmark tests how well the fastest scheduling options scale with different amounts of threads. For the benchmark n was set to 729.

The different scheduling options used in the first phase are:

- Auto
- Static

- Static, Dynamic, Guided, all with different chunk sizes of: 1, 2, 4, 8, 16, 32, 64

The fastest scheduling options for the critical sections are then run with 1, 2, 4, 6, 8, 12 and 16 threads during the second phase of the benchmark.

Like stated in the introduction, both benchmark phases are executed on the Cirrus back end with exclusive access to one node. Every scheduling option in phase one and every amount of threads in phase two were executed 100 times and the average and median walltime—in seconds—were measured with the timing routine `omp_get_wtime`, provided by OpenMP (see OpenMP Architecture Review Board, 2015, Chapter 3.4.1). The average walltime was used as the decisive criteria for execution speed. The mean was used as the secondary criteria.

3. Results

Table 1 lists the average and median walltime in seconds for the execution of the critical sections, determined in phase one of the benchmark.

Sequential is not a schedule. It represents execution time of the critical section in a sequential, not parallelized manner.

One can see, that for the first—the symmetric—critical section the schedules do not differentiate much concerning the execution time. Especially the Auto, Dynamic, n and Guided, n scheduling options are all between 0.48 and 0.52 seconds of average execution time. Static, n performs slightly worse, all options having an average execution time between 0.51 and 0.56 seconds, except Static, 64, which performs worse with an average execution time of 0.62 seconds. The Static scheduling option, which just splits the iterations in #threads (in this case four) approximately equal chunks, performs the worst with an average execution time of 0.83 seconds (see Mark Bull, 2019).

The best scheduling option for the first critical section is Dynamic, 16, with an average and median execution time of 0.48 seconds.

The schedules differ much more in execution time for the second critical section, compared to the first one.

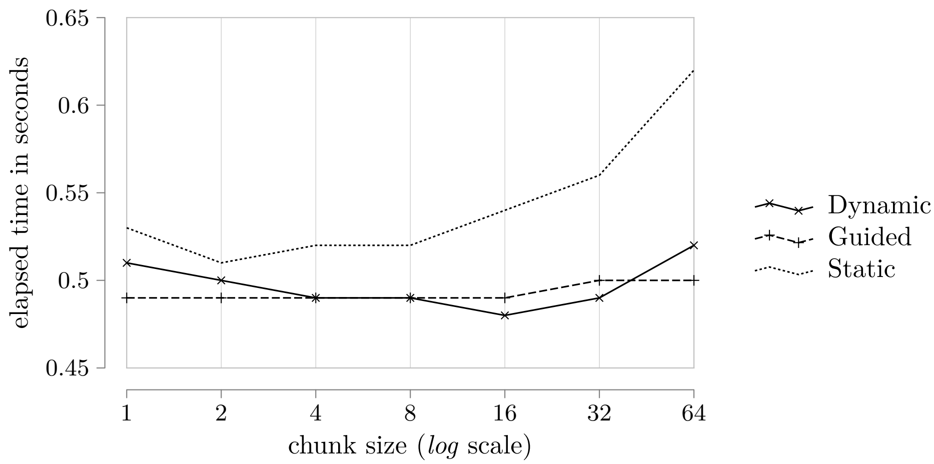
The difference between Dynamic, 8 (fastest) and Static (slowest) is nearly 4 seconds, which makes Static approximately 2.8 times slower than Dynamic, 8. For comparison, the slowest scheduling option for the first critical section is just approximately 1.7 times slower than the fastest scheduling option.

Guided, n and Auto, which were close to fastest in the first critical section, perform worse on the section critical section. All are approximately 2.4 times slower than Dynamic, 8.

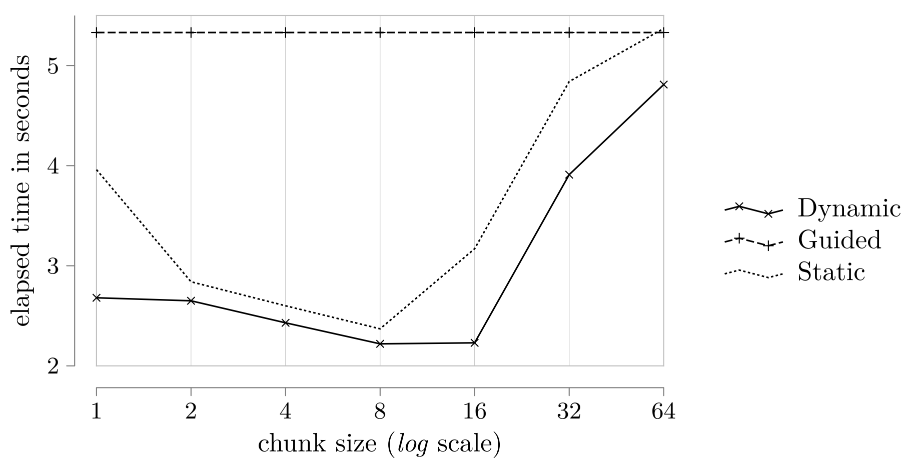
Static, n fluctuates the most with different chunk size n. Static, 8, with 2.37 seconds average execution time, is the third fastest scheduling option, while Static, 64 is the second slowest scheduling option with 5.37 seconds average execution time.

schedule	critical section 1		critical section 2	
	mean	median	mean	median
Sequential	1.62	1.61	8.57	8.56
Auto	0.49	0.48	5.32	5.32
Static	0.83	0.84	6.18	6.19
Dynamic, 1	0.51	0.51	2.68	2.57
Dynamic, 2	0.50	0.49	2.65	2.57
Dynamic, 4	0.49	0.49	2.43	2.39
Dynamic, 8	0.49	0.48	**2.22**	**2.22**
Dynamic, 16	**0.48**	**0.48**	2.23	2.23
Dynamic, 32	0.49	0.48	3.91	3.91
Dynamic, 64	0.52	0.51	4.81	4.81
Guided, 1	0.49	0.49	5.33	5.34
Guided, 2	0.49	0.48	5.33	5.33
Guided, 4	0.49	0.49	5.33	5.33
Guided, 8	0.49	0.48	5.33	5.33
Guided, 16	0.49	0.48	5.33	5.33
Guided, 32	0.50	0.49	5.33	5.33
Guided, 64	0.50	0.49	5.33	5.33
Static, 1	0.53	0.53	3.96	3.93
Static, 2	0.51	0.51	2.84	2.81
Static, 4	0.52	0.52	2.60	2.57
Static, 8	0.52	0.52	2.37	2.37
Static, 16	0.54	0.53	3.17	3.18
Static, 32	0.56	0.56	4.84	4.84
Static, 64	0.62	0.63	5.37	5.38

Table 1: Results of phase one of the benchmark. Displayed are average and median walltime in seconds for every scheduling option for both critical sections. The fastest scheduling options are marked with a bold font-weight.



(a) Critical section 1.



(b) Critical section 2.

Figure 1: Plots on how the chunk size clause changes the execution speed of the Dynamic, Guided and Static scheduling options, for both critical sections.

#threads	critical section 1		critical section 2	
	mean	median	mean	median
1	1.87	1.87	8.59	8.59
2	0.94	0.93	4.30	4.30
4	0.48	0.48	2.22	2.22
6	0.34	0.34	2.08	2.08
8	0.26	0.26	2.09	2.10
12	0.19	0.18	2.08	2.08
16	0.15	0.14	2.07	2.07

Table 2: Results of phase two of the benchmark. Displayed are average and median walltime in seconds for the fastest scheduling options from phase one, for each critical section, executed with different amounts of threads.

The fluctuation of the average execution time, based on different chunk sizes can be seen in Figure 1.

During the second phase of the benchmark the two scheduling options resulting in the fastest average execution time were tested with different amounts of threads. The fastest scheduling option for the first critical section was Dynamic, 16. For the second critical section Dynamic, 8 resulted in the fastest average execution time.

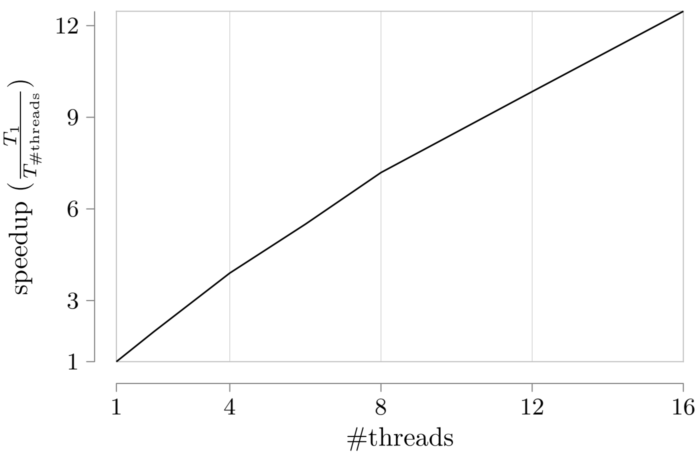
Table 2 lists the average and median execution time in seconds for both scheduling options when run with 1, 2, 4, 6, 8, 12 and 16 threads.

Figure 2 displays how much using more threads gains in exectution speed, compared to using just one thread (sequential execution). While for the first critical section the speedup of using more threads grows linear, the execution time for the second critical section stops being faster after 6 threads (see Table 2, Figure 2).

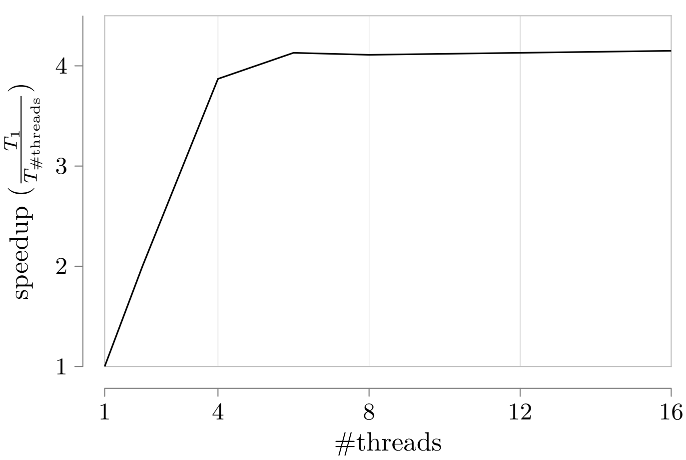
4. Discussion

There are several statements to make about why some scheduling options outperform each other.

For the first critical section only Static and Static, n perform worse than the others. Guided, n has approximately the same performance as Dynamic, n (also it is more constant with different chunk sizes). This suggests, that the reason why Static and Static, n perform worse must lie in the later iterations. This could have two reasons: (i) longer indexing of $B(i,j)$; (ii) $\cos(x)$ takes longer to compute for bigger x (see Equation 1). Problem (ii) could easily be solved by setting $B(i,j) = B(i,j) \mod 2\pi$, since if $x \equiv y (\mathrm{mod}\ 2\pi)$, then $\cos(x) = \cos(y)$ (follows from the fact that $\cos(x)$ has 2π as its period) (see e.g. Romanowski, 2014).



(a) Critical section 1.



(b) Critical section 2.

Figure 2: Plots on how the execution speed varies with the amount of threads. The plots show how much faster more threads are, compared to just one thread.

While it is obvious that Static, n with big n performs poorly on big chunks (the first threads will run very long—see Chapter 2)

5. Conclusion

References

EPCC. Cirrus, 2019. URL `https://www.cirrus.ac.uk`.

Intel. Intel Fortran Compiler 17.0 for Linux, 2016. URL `https://software.intel.com/en-us/articles/intel-fortran-compiler-170-for-linux-release-notes-for-intel-parallel-studio-xe-2017`.

Mark Bull. Threaded Programming Lecture 4: Work sharing directives, 2019.

OpenMP Architecture Review Board. OpenMP application program interface version 4.5, 2015. URL `https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf`.

Perry Romanowski. Trigonometry, 2014.