University of Basel

Exploring Performance of Loops with Dependencies versus Tasks with Dependencies in Multi-threaded applications using OpenMP

Master project

Natural Science Faculty of the University of Basel
Department of Mathematics and Computer Science
High Performance Computing Group

Reviewer: Prof. Dr. Florina M. Ciorba
Second reviewer: Ali Omar Abdelazim Mohammed

Reto Schmid
reto.schmid@unibas.ch
10-056-596

May 20, 2020

Table of Contents

1 Summary

Algorithms used in scientific and other applications mostly have some sort of dependencies, preventing embarrassingly parallel implementations. Among others, loop dependencies pose a problem which occurs quite frequently. To maximise performance of such problems while exploiting available parallelism with OpenMP, we compare different scheduling techniques for For-loops. Additionally, the same tests are made with OpenMP Tasks to see whether For-loops or Tasks leads to better performance results. More precisely, we conduct a set of experiments to assess and compare eleven scheduling techniques. At the time this report is written, there are still things to be adjusted, so no final conclusion can be made yet. The work continues while this results represent the current state of work.

2 Introduction

Algorithms used in scientific and other applications mostly have some sort of dependencies, preventing embarrassingly parallel implementations. The goal of this project is to explore the performance of loops with cross-iteration dependencies (DOACROSS loops) in OpenMp For-loops and compare them to equivalent implementations with OpenMP Tasks. Due to the small extent of this project, the tests were made exemplary on the Fibonacci algorithm and on parts of the Parallel Research Kernels (PRK)[1]. Three parts of the P2P kernel are tested in this project: *p2p-doacross-openmp.cc*, *p2p-hyperplane-openmp.cc* and *p2p-tasks-openmp.cc*. The *p2p-doacross-openmp.cc* implementation is additionally tested with eleven different scheduling techniques. The point of interest is which implementation gives better performance for cross-iteration dependencies.

[1] https://github.com/jeffhammond/PRK

3 Background

A brief background on some important topics for this project is presented in this section, including a study on DOACROSS loops [4] and a master thesis about scheduling algorithms in an OpenMP library [5].

3.1 DOACROSS loops

Unlike DOALL loops (i.e. loops without cross-iteration dependencies[4]), DOACROSS loops can have data or control dependencies crossing iteration boundaries[4]. The more interesting type of dependencies are data dependencies, since control dependencies can be detected statically by a control dependence analysis. There are three types of data dependencies:

- Flow dependencies
- Anti dependencies
- Output dependencies

Many loops with only anti- and output dependencies can be transformed into DOALL loops by a good optimising compiler [4]. Ding-Kai Chen 1991 states that directives from programmers could help to fully exploit parallelism on DOACROSS loops and that the loss of parallelism after serialising DOACROSS loops is quite significant.

3.2 Scheduling techniques

Parallelising loops can have very positive impacts on the performance of programs. Distributing chunks of iterations over different processing units (PUs) increases the performance of the program. However, not all iterations of a loop necessarily contain the same amount of computational work, which can lead to load imbalance. If one PU finishes before others and is idling, its computational power is partially wasted. To minimise wasted resources, we need different scheduling techniques for different applications. However, scheduling involves overhead which again can lead to bad performance. That said, there is always a trade-off

between load balancing and scheduling overhead. Depending on the application, different scheduling techniques may lead to vastly different performances.
Common variable used in the following description of different scheduling techniques are

Variable	Description
N	Number of iterations.
P	Number of PUs.
C	The chunk size.
r	Number of remaining iterations.

3.2.1 Static Chunking

Static Chunking (*static*) is a static loop scheduling technique, where a loop is decomposed into P equal sized chunks of iterations. The chunk size is therefore

$$C = \frac{N}{P} \tag{3.1}$$

This scheduling techniques leads to close to zero overhead.

3.2.2 Self Scheduling

Self Scheduling (*dynamic*) is on the other side of the trade-off scale of loop scheduling techniques. It always assigns a single new iteration to an idling PU. The chunk size is

$$C = 1 \tag{3.2}$$

Intuitively, Self Scheduling provides the best possible load balancing while producing the biggest scheduling overhead.

3.2.2.1 Guided Self Scheduling

Guided Self Scheduling (*guided*) tries to reduce the overhead time of Self Scheduling by assigning decreasing chunk sizes to the PUs. It tries to reduce overhead with less chunks while still providing good load balance. The chunk size is

$$C_i = \lceil \frac{r_i}{P} \rceil \tag{3.3}$$

where r_i is the remaining number of iterations and $r_1 = N$.

3.2.3 Trapezoid Self-Scheduling

Trapezoid Self-Scheduling (*trapezoidal*) wants to extract the advantage of GSS and at the same time provide a simple linear function for decreasing chunk sizes. Furthermore, it takes two inputs from the user which specify the size of the first chunk, f, and last chunk l.[5] The chunk size is

$$A = \lceil \frac{2N}{f+l} \rceil, \tag{3.4}$$

$$\delta = \frac{f-1}{A-1}, \tag{3.5}$$

$$C(1) = f, \tag{3.6}$$

$$C(t) = C(t-1) - \delta. \tag{3.7}$$

where t is the number of the current scheduling operation and A is the number of total scheduling operations.

3.2.4 Factoring and Adaptive Factoring Techniques

The factoring techniques are more sophisticated versions of Guided Self-Scheduling. They also implement decreasing chunk sizes for better load balancing while trying to be more resistant to iteration execution time variance. To achieve this, batches of iterations are scheduled with different chunk sizes. Adaptive loop scheduling methods use (additional) information obtained during runtime for their scheduling decisions. More mathematical background and further explanations for all scheduling techniques used in this project can be found in [5].

3.3 OpenMP

The OpenMP ARB (Architecture Review Boards) mission is to standardize directive-based multi-language high-level parallelism that is performant, productive and portable. Jointly defined by a group of major computer hardware and software vendors and major parallel computing user facilities, the OpenMP API is a portable, scalable model that gives shared-memory parallel programmers a simple and flexible interface for developing parallel applications on platforms ranging from embedded systems and accelerator devices to multicore systems and shared-memory systems[3].

4

Experimental Setup

4.1 MiniHPC

All of the results of this project were acquired using the miniHPC cluster[2] from the University of Basel. The cluster has two types of nodes, Intel Xeon nodes (Xeon) and Intel Xeon Phi Knights Landing (KNL) nodes. One Xeon node is used for login, one fore storage and the remaining 22 for computing. The four Intel Xeon Phi nodes are solely computing nodes. All nodes are interconnected through two different types of interconnection networks. The first network is an Ethernet network with 10 Gbit/s speed, reserved for users and administrators access. The second network is the fastest network, an Intel Omni-Path network with 100 Gbit/s speed, reserved for the high-speed communication between the computing nodes. The topology of this second network interconnects the 28 nodes (24 Xeons and 4 KNLs) of the miniHPC cluster via a two-level fat-tree topology.[2]

4.2 Fibonacci series application

The fibonacci series implementations are straight forward. Due to the sequential nature of the algorithm, the use of parallel computation is not expected to bring any speedup. It was done for exemplary reason and so see how the different schedules work. The doacross loop version works with the *ordered* clause to take account of the data dependencies. The temporary fibonacci numbers are stored in a global array, so other threads can access the data. All version of the algorithm perform the task of calculating the 20000th element of the fibonacci series. This is close to the limit which is given by the high numbers resulting from the calculations exceding the storage capacity of 128-bit doubles.

4.2.1 Fibonacci with OpenMP for loop

```
long double* a=(long double*) malloc (N*sizeof(long double));
#pragma omp parallel for schedule(runtime) ordered (1)
for(int i = 0;i<N;++i){
    if(i<2){
```

[2] https://hpc.dmi.unibas.ch/HPC/miniHPC.html

```
            a[i] = i;
            #pragma omp ordered depend(source)
      } else {
            #pragma omp ordered depend(sink:i-1) depend(sink:i-2)
            {
                  a[i] = a[i-1] + a[i-2];
                  #pragma omp ordered depend(source)
            }
      }
}
```

4.2.2 Fibonacci with OpenMP Tasks

```
long double fibonacci(int n){
     if (n<2){
      a[n] = n;
      return n;
     } else {
      #pragma omp task
      {
            if(a[n-2]== 0){
                  a[n-2]=fibonacci(n-2);
            }
            if(a[n-1]== 0){
                  a[n-1]=fibonacci(n-1);
            }
      }
      #pragma omp taskwait
      a[n] = a[n-1]+a[n-2];
      return a[n];
     }
}
```

4.3 Parallel Research Kernels (PRK)

The second part of the test was made with a number of kernel operations, called Parallel Research Kernels. Namely parts of the P2P kernel implemented in C++11 from [1] were used:

- *p2p-doacross-openmp.cc* 2D-matrix calculations in for loops with cross iteration data dependencies
- *p2p-hyperplane-openmp.cc* 2D-matrix calculations in for loops in a hyperplane fashion with cross iteration data dependencies
- *p2p-tasks-openmp.cc* 2D-matrix calculations in OpenMP tasks with cross tasks data dependencies

For compilation the enclosed Makefile with the necessary adjustments for the intel compiler was used. Different compiler flags were used depending on if the code was run on the Xeon or the KNL nodes according to the directions in the *./commons/make.defs.intel* file.

4.4 Design of Experiments

Factors	**Values**	**Properties**
Applications	Fibonacci Series	N = 20000

	PRK P2P	N = 4000
		Chunksize = 1

Thread level load balancing	*static*	static scheduling
	dynamic, guided, trapezoidal, fac2, af, af_a, awf_b, awf_c, awf_d, awf_e	dynamic scheduling
miniHPC computing system	Xeon node without hyperthreading	Intel Xeon node; 20 threads
	Xeon node with hypertreading	Intel Xeon node; 40 threads
	KNL node without hyperthreading	Intel Xeon Phi Knights Landing (KNL) node; 64 threads
	KNL node with hyperthreading	Intel Xeon Phi Knights Landing (KNL) node; 256 threads

Table 4.1: Details used in the design of the experiments for performance analysis

There are 2 different applications which were tested in 4 different environmental settings. DOACROSS for loops were additionally tested multiple times with different scheduling techniques. More details are included in table 4.1

4.5 Visualisation

To be able to visualise the assignment of the calculation steps to the calculating thread, simple print statements inside the loops and tasks were used:

```
printf("Index:[%d,%d], Time: %f, CalcThread:%d\n", i, j, prk::wtime()-
    pipeline_time, omp_get_thread_num());
```

To get the plots from the resulting entries, a Python script was used. To compare execution times the print statements were omitted for obvious reasons.

5 Performance Results And Discussion

5.1 Compiler optimiser flag: -O0 vs -O2

To make sure the compiler doesn't interfere with the tests by optimising things, the standard optimisation flag (-O2) was compared to no compiler optimisation. This was made with the PRK codes. The results can be seen in figure 5.1. While the absolute values are of no interest at this point, we can see that there is no influence of the -O2 optimisation across the different scheduling techniques. Compiled with Intel compiler version 2019 and the -O2 flag, every tested configuration is a bit faster than the -O0 counterpart, but the relation stays about the same.

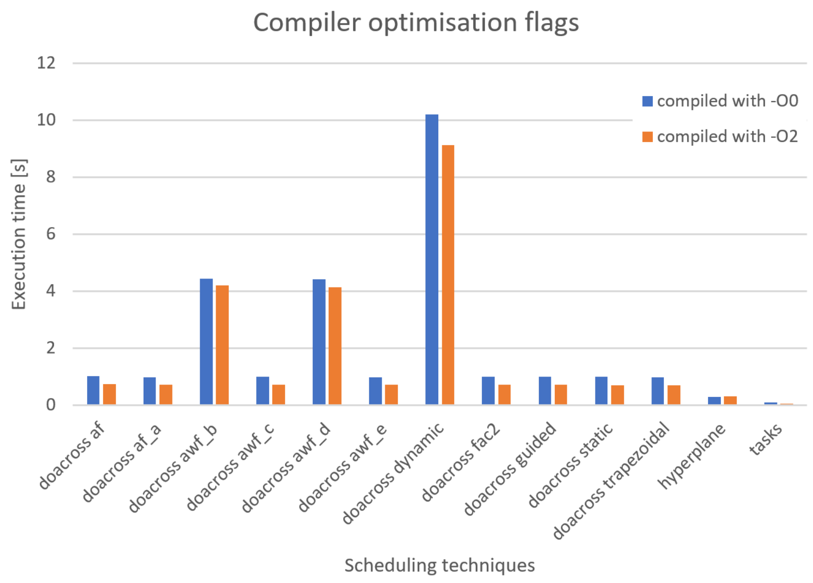

Figure 5.1: Comparison of execution times with Intel compiler version 2019 and optimisation flags O0 and O2

5.2 Fibonacci

The execution time measurements for the Fibonacci algorithm show some huge differences between the doacross for loop version and the tasks version. While the 20000th element of the fibonacci series is computed in milliseconds with for loops, it takes some dozen seconds with tasks. An interesting fact is also, that the task version is very inconsistent in its time consumption. Execution times vary between 10s and 32s.

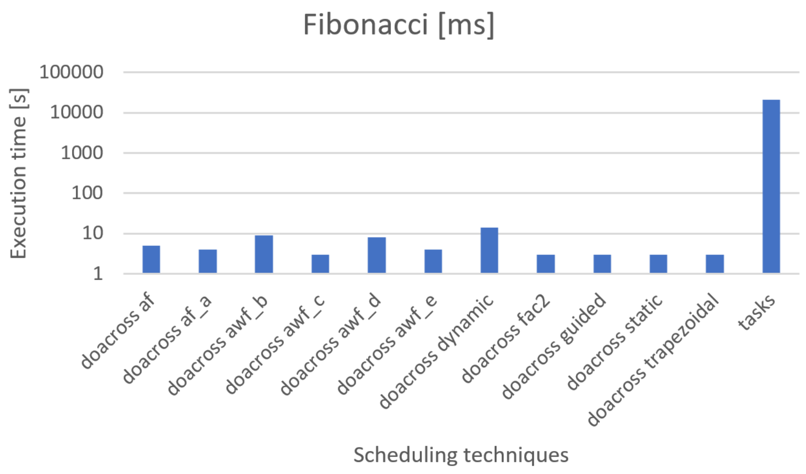

Figure 5.2: Execution time of fibonacci(20000) tasks compared to for loops

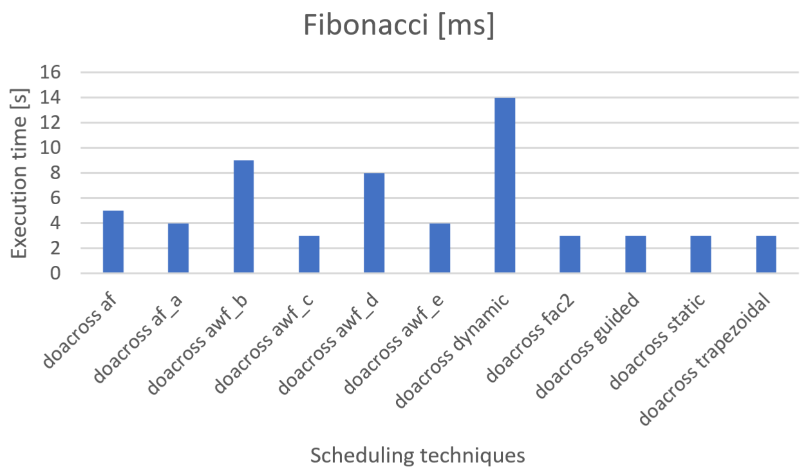

Figure 5.3: Execution time of fibonacci(20000) for loops with different scheduling techniques only

One of the things that need some refinement in this project is the automated plotting of the load assignment. With 20000 calculations to be plotted, it is necessary to display chunks of data assigned to a given thread rather than having a dot for every single calculation. Otherwise there can't be seen anything because of indistinguishable data points. This is the

reason we omit those plots at this time.
From the tasks output file can be seen that huge chunks of calculations are done by the same thread. This is surprising, since every thread should have the same chance to get to calculate the next fibonacci iteration when the previous is done.

5.3 Parallel Research Kernels

The measurements for all PRK environment setups were taken with an arbitrary grid size of 4000 x 4000. The execution times are averaged over five complete calculations. This seems to take a reasonable amount of time for comparison. All three codes (doacross, hyperplane and tasks) can be provided with another parameter (pair) called chunksize. At the time of this report, it's still to clarify if this parameter does the very same for all of the three codes. It defines the size of blocks of calculations which are then assigned to tasks, for instance. Since this is what we're trying to achieve with different load balancing techniques, it was at this point set to 1 at both dimension of the grid for all calculations.

5.3.1 PRK on Xeon node without hypertreading

The execution time measurements on a Xeon node **without** hyperthreading can be seen on figure 5.4. The task version didn't get a result because it wasn't finished with the calculations even after 30 minutes and got cancelled. On the other hand, the hyperplane code was more than 10 times faster than the fastest doacross version. The scheduling techniques *awf_b, awf_d* and *dynamic* seem to perform worse than the other scheduling techniques.

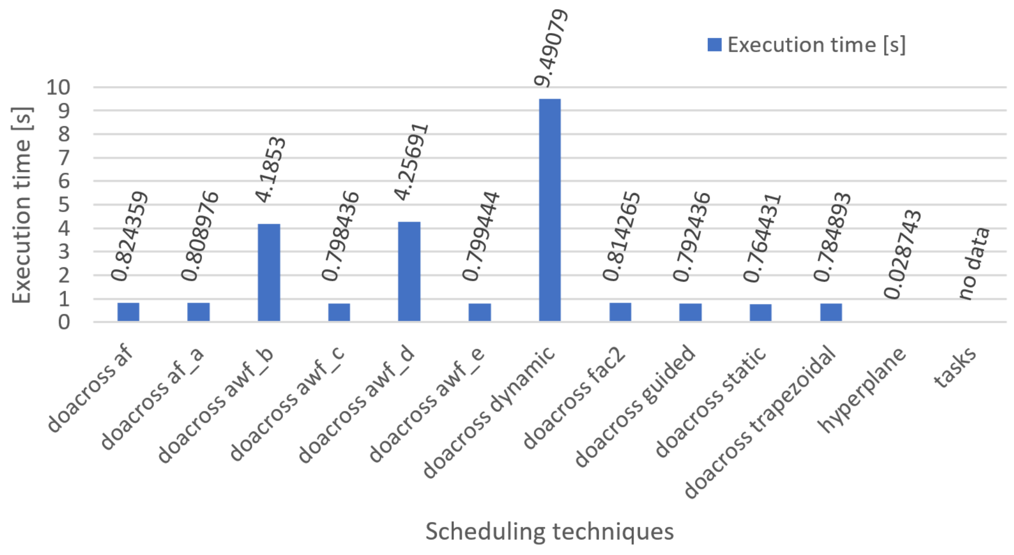

Figure 5.4: Execution times on Xeon node without hyperthreading

5.3.2 PRK on Xeon node with hyperthreading

The execution time measurements on a Xeon node **with** hyperthreading can be seen on figure 5.5. The task version again didn't finish with the calculations in under 30 minutes, and is therefore ignored. The overall picture is the same, the scheduling techniques *awf_b, awf_d* and *dynamic* seem to perform worse than the others, while the hyperplane code performs best by far. The results for the doacross versions are a bit worse than without hyperthreading, while staying roughly the same for the hyperplane code.

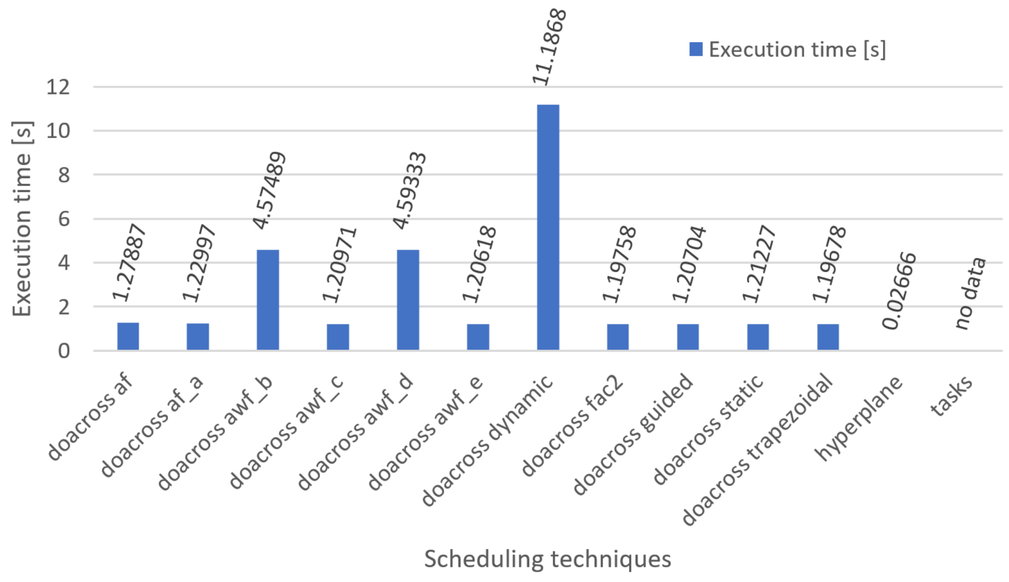

Figure 5.5: Execution times on Xeon node with hyperthreading

5.3.3 PRK on KNL node without hyperthreading

The execution time measurements on a KNL node **without** hyperthreading can be seen on figure 5.6. Once more, there is no value for the task version. The overall performance on all of the other codes is substantially worse for the KNL node.

5.3.4 PRK on KNL node with hyperthreading

The execution time measurements on a KNL node **with** hyperthreading can be seen on figure 5.7. Unsurprising, there is yet again no value for the task version. With some of the scheduling techniques, the results fall a bit out of line compared to the other measurements. *Dynamic* scheduling still performs worst, and *awf_b, awf_d* worse than others. But with hyperthreading on the KNL node, also *af* and *af_a* perform significantly worse compared to the other measurements. The overall performance is also worse with hyperthreading, similar to the measurements on the Xeon node.

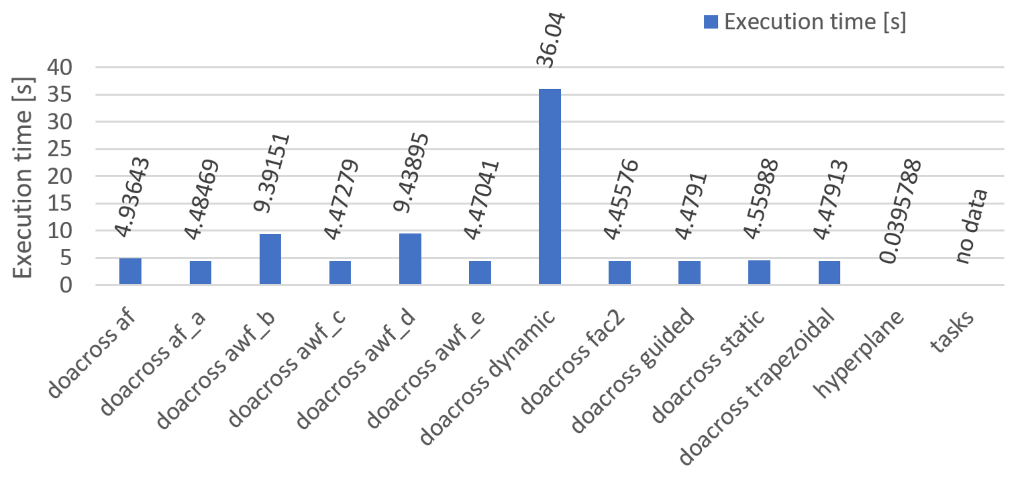

Figure 5.6: Execution times on KNL node without hyperthreading

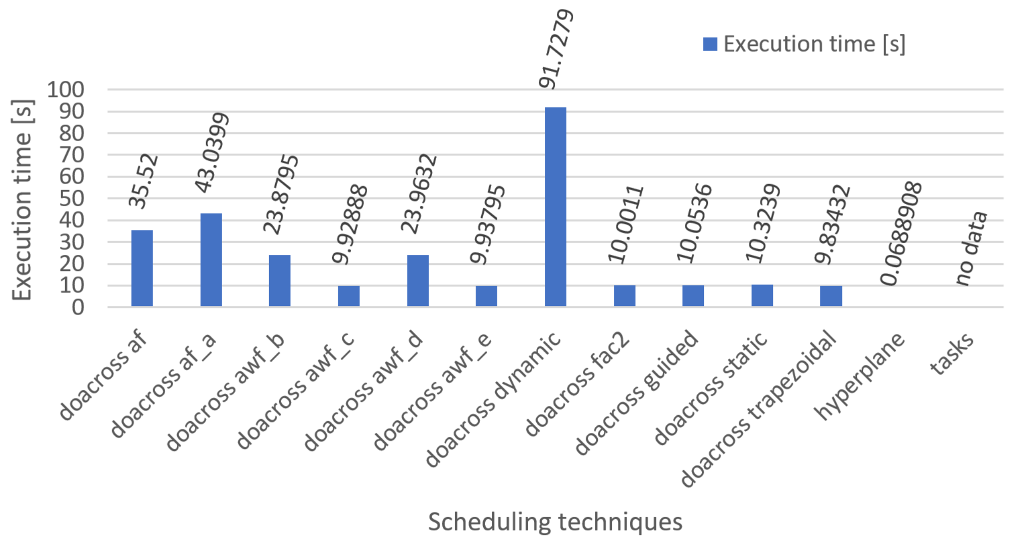

Figure 5.7: Execution times on KNL node with hyperthreading

5.3.5 Discussion

The hyperplane code performs best independent of the setup. This is most likely due to the fact that the hyperplane fashioned code uses OpenMP SIMD instructions while chunksize is set to 1. To really have comparable results, more tests with other chunksize values have to be and will be made. A further question is probably, why SIMD instructions work particularly well on this type of problem. But since the SIMD implementation is restricted on chunksize equals 1, it's probably not that interesting at all. OpenMP Tasks perform really bad or are erroneous. No further investigations have been made at this point. It is assumed that creating $4000 \cdot 4000 = 16 \cdot 10^6$ tasks (one for every single calculation) with cross task dependencies leads to so much overhead, it's not feasible to compute anymore. Having 40 instead of 20 threads on a Xeon node and 256 instead of 64 threads on a KNL node seems to lead to much more overhead. This is assumed to also be a problem of the small chunksize value. Every single calculation is assigned to a thread. This seems to be quite inefficient. Due to this, we could also expect better performance from increasing the chunksize value for any configuration. Since the KNL node operates with more threads than the Xeon node and the assignment to the threads seems to cause a lot of overhead compared to the actual calculation, this is probably the explanation here as well. With bigger values for chunksize, this relation will probably change. Overall, a lot of the performance differences are likely to come from massive overhead in scheduling due to the smallest possible chunksize, therefore additional experiments will give much more insights.

6 Conclusion and Upcoming Work

With the current setup, the following observations can be made:

a. The hyperplane code performs better than any scheduling techniques or tasks.

b. OpenMP tasks are erroneous or perform really bad.

c. While hyperthreading is activated, all scheduling techniques perform worse than without hyperthreading.

d. The overall performance on the Xeon node is better than on the KNL node

e. Independent of the hardware configuration, *dynamic* scheduling seems to perform worst for this problem. *Awf_b* and *awf_d* perform worse than the other scheduling techniques, *dynamic* excluded. All other scheduling techniques perform roughly at the same level, the exception being *af* and *af_a* with KNL hyperthreading.

At this point, with the results at hand, more measurements with different chunksize values is the obvious next step. The hyperplane code will also work with for loops for any other chunksize value than 1 and therefore would also be interesting to test with different scheduling techniques.
The second thing to be done is a working visualisation of the task assignments to the threads to increase the understanding of what the different scheduling techniques actually do. One way will be with more sophisticated print statements and a graphical representation using python.

Bibliography

[1] Parallel research kernels. URL https://github.com/ParRes/Kernels.

[2] Minihpc, small but modern hpc. URL https://hpc.dmi.unibas.ch/HPC/miniHPC.html.

[3] Openmp - about us. URL https://www.openmp.org/about/about-us/.

[4] Pen-Chung Yew Ding-Kai Chen. An empirical study on doacross loops. 1991.

[5] Arkan Yilmaz. Implementation of scheduling algorithms in an openmp runtime library, 2019.

Declaration on Scientific Integrity
Erklärung zur wissenschaftlichen Redlichkeit

includes Declaration on Plagiarism and Fraud
beinhaltet Erklärung zu Plagiat und Betrug

Author — Autor
Reto Schmid

Matriculation number — Matrikelnummer
10-056-596

Title of work — Titel der Arbeit
Exploring Performance of Loops with Dependencies versus Tasks with Dependencies in Multi-threaded applications using OpenMP

Type of work — Typ der Arbeit
Master project

Declaration — Erklärung
I hereby declare that this submission is my own work and that I have fully acknowledged the assistance received in completing this work and that it contains no material that has not been formally acknowledged. I have mentioned all source materials used and have cited these in accordance with recognised scientific rules.

Hiermit erkläre ich, dass mir bei der Abfassung dieser Arbeit nur die darin angegebene Hilfe zuteil wurde und dass ich sie nur mit den in der Arbeit angegebenen Hilfsmitteln verfasst habe. Ich habe sämtliche verwendeten Quellen erwähnt und gemäss anerkannten wissenschaftlichen Regeln zitiert.

Basel, May 20, 2020

[illegible]