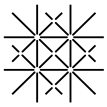

University
of Basel

Evaluation and Analysis of Dynamic Loop Scheduling in OpenMP

Master Thesis

Natural Science Faculty of the University of Basel
Department of Mathematics and Computer Science
High Performance Computing
hpc.dmi.unibas.ch

Examiner: Prof. Dr. Florina M. Ciorba
Supervisor: Ali Omar Abdelazim Mohammed, MSc.

Patrick Buder
p.buder@unibas.ch
09-062-480

30.10.2017

Acknowledgments

I would like to say thanks to all the people who helped me along the way of creating this thesis. First of all I thank Ali Omar Abdelazim Mohammed supervising me during my thesis. He was always available for questions and helped me to grasp many new concepts. Any time I got stuck during the coding or experiments, he we sat together and worked it out.

Next I'd like to say my thanks to Prof. Dr. Florina M. Ciorba for offering this interesting topic as a master thesis and accepting me into the High Performance Computing group at the University of Basel. She helped guide the direction of this thesis during our many meetings and deepened my knowledge of the topics covered here.

Furthermore I'd like to thank my many friends who listened to my sometimes rather basic questions and answered them with patience.

Finally I want to say my thanks to Pedro H. Penna, who helped me get started with his implementation of libgomp, the gnu OpenMP runtime library, on which the implementation of our dynamic loop scheduling methods is built.

Abstract

Loops are the main source of parallelism in most scientific applications. Many of these loops are used to execute the same operations on multiple data elements. To reduce execution time, this can be done in parallel on different processing units. Distributing the iterations of a parallel loop evenly across available processing units is the job of loop schedulers. These schedulers employ scheduling methods to calculate and assign chunks of work to processing units in such a way as to achieve a balanced load execution among them. Their goal hereby is to minimize the loop execution times. If the execution time of single loop iterations varies greatly, dynamic loop scheduling methods should be deployed to ensure a good workload balance.

The OpenMP specifications currently offer three different scheduling methods. These three methods do not always sufficiently express the parallelism in an application, which can lead to a suboptimal exploitation of the hardware and result in a loss of performance. In this thesis, the GNU OpenMP runtime library libgomp was extended by six additional dynamic loop scheduling methods. The resulting eight different dynamic loop scheduling methods and one static loop scheduling method are evaluated on their performance on multiple benchmarks from five different benchmark suites. The results of the evaluation show no clear superiority of one loop scheduling method above the others but rather confirm that different loop scheduling methods are needed for different applications.

Acronyms

BOLD Bold Strategy.

DLS Dynamic Loop Scheduling.

FAC Factoring.

FSC Fixed Size Chunking.

GCC GNU Compiler Collection.

GSS Guided Self Scheduling.

libgomp GNU Offloading and Multi Processing Runtime Library.

NAS NAS Parallel Benchmark Suite.

OMPSCR OpenMP Source Code Repository.

OpenMP Open Multi-Processing.

PU Processing Unit.

SPEC OMP SPEC OpenMP Benchmark Suite.

SRR Smart Rount-Robin.

SS Self Scheduling.

TAPER Taper Strategy.

TSS Trapezoid Self Scheduling.

WF Weighted Factoring.

Table of Contents

1 Introduction

was soll das denn heissen?!

Modern computers feature a growing amount of Processing Unit (PU)s, ever increasing the demand for more parallelism. In scientific applications, this parallelism can often be exposed on loops. Inside these applications, the parallelism can then be expressed further. This expression can be supported by the Open Multi-Processing (OpenMP) specifications, which contain compiler directives, library routines and environment variables for parallelization of C, C++ and Fortran code [1]. When using the implementation of OpenMP in one of many compilers, only a few lines of additional code have to be added to create a parallel loop. The OpenMP implementation GNU Offloading and Multi Processing Runtime Library (libgomp) will be used throughout this thesis.

There are multiple ways to distribute the workload of one parallel loop across available PUs. These different ways are called loop scheduling methods. OpenMP currently contains three different methods to schedule loops. The use of these three methods does not always lead to a perfect exploitation of the underlying hardware, resulting in less than optimal performance. Inefficient exploitation of hardware can be the result of heavy variance in loop iteration execution time. Applications which exhibit such variance inside their loops are also called irregular applications. Dynamic Loop Scheduling (DLS) methods try to work around the variance in irregular applications by assigning iterations to PUs during runtime of an application. This usually results in a better balanced load across all PUs than if static scheduling is used. Static loop scheduling assigns equal amounts of loop iterations to each core disregarding differences between iteration execution times or core speeds. It generally leads to best performance on regular applications, which do not exhibit heavy variance between loop iterations. In DLS, the amount of iterations each core gets assigned at a time varies between the different DLS algorithm. The most basic DLS method, Self Scheduling (SS), assigns one iteration at the time and guarantees the best possible load balance this way.

Both static scheduling and SS are already available in all official OpenMP implementations. The third method in OpenMP, Guided Self Scheduling (GSS), tries to strike a balance between the other two methods. It assigns chunks in descending chunk sizes to PUs.

A wide range of other DLS exist, yet OpenMP still only specifies these three. Some imple-

mentations of OpenMP were already extended by additional scheduling methods, including libgomp. More about these methods can be found in chapter 3. In this thesis we expand the supported scheduling methods of libgomp by implementing six additional DLS and evaluate their performance against the existing methods.
The following chapters are organized as: In chapter 2 a quick overview of parallelism, loop scheduling and OpenMP is given. Then each loop scheduling algorithm used in this thesis is explained with appropriate formulas. In chapter 3 we list previous work on extending the OpenMP loop scheduler by two different groups. chapter 4 contains an explanation as to the whys and hows of this thesis. Why and how we implemented the DLS algorithms. chapter 5 gives a detailed explanation how to extend libgomp and the benchmarks. It is meant to give proof, that the methods were implemented correctly and will make further efforts to modify libgomp easier. chapter 6 is about the benchmarks and experiments that were run to test and measure the implementation. In chapter 7 a quick summary over the whole thesis is given and possible future work in this topics is covered.

2
Background

In this chapter the background knowledge needed to understand this thesis is provided. It starts with a general explanation of parallelism and scheduling, then it transitions into detailed descriptions of the different loop scheduling methods. At the end, a quick overview over OpenMP is given.

2.1 Parallelism

Parallelism in computer science is the execution of two or more operations at the same time. An operation in this context would be an arithmetic or logical operation on some data, for example a simple addition of two numbers. On modern central processing units (CPUs) there are several cores or PUs. Each PU usually does one operation at the time. Two PUs executing one operation each at the same time would be called a parallel execution. In this thesis we focus on the parallel execution of loops. Each PU will receive a portion of iterations of the parallelized loop to compute.

2.2 Scheduling of Parallel Loops

Scheduling denotes the distribution in space and over time of a workload among the available processing cores. There are different ways to categorize scheduling strategies and methods. One of them distinguishes between static and dynamic scheduling. Another one takes adaptiveness into account. There are also scheduling strategies which take weights for different PU speeds into account. The main part of this thesis is concentrated on loop scheduling. Loops are specific parts in an application, where the same instructions are executed multiple times. Most of the time, the difference in execution time between the loop iterations is small. Still, sometimes bigger differences can occur. For example when different inputs are used for different iterations or if there are many conditional statements inside of a single loop. In loop scheduling, the main goal is to distribute the loop iterations across all available PUs.
Static loop scheduling is the most intuitive, where the iterations of a loop are evenly distributed among all PUs. Every PU obtains an equal number of iterations. In DLS, the PUs obtain a certain amount of iterations whenever they become available. When they finish

Table 2.1: Common variable names

Variable	Description
Cs	Chunk size
P	Number of Processes
N	Number of Iterations
R	Remaining Iterations
μ	Mean execution time of the iterations
σ	Standard deviation of the execution time of the iterations
h	Overhead time

their assigned iterations, they acquire more. The amount of iterations assigned each time is regulated by different methods, a few of which are more closely explained in this chapter.
Weighted loop scheduling assigns different weights to each PU according to its processing speed. A faster PU will get more weight assigned and therefore handle more loop iterations, since it is assumed to be able to handle more work in the same amount of time as a slower PU.
Adaptive loop scheduling is the newest and most advanced of the scheduling categories. It dynamically adapts the amount of iterations one PU receives during the execution of the program according to multiple factors. The amount of time it took to compute the most recently finished iteration can, for example, be one of those factors.
Dynamic loop scheduling has the advantage of balancing the load when loop iteration execution times have high variability. One example would be an integer sorting, where the amount of calculations in each iteration depend on the input. Adaptive methods are being used with irregular applications, where the mean and standard deviation of iteration execution times is unknown and iterations have a high variability.
Scheduling methods try to optimize the two parameters load balance and scheduling overhead. We can use these to describe the scheduling methods. Each method strikes a balance between the two parameters. The two extremes would be the static and SS methods. The static method has the least scheduling overhead of all methods, while the SS method usually balances the load as optimally. In turn, both of them usually achieve bad results for irregular and regular applications respectively.

2.2.1 Static Scheduling Methods

Static scheduling denotes a way of scheduling, where the allocation of tasks to PUs is known beforehand. This means, a scheduling method, which precomputes a perfect task to PU alignment, would also be called static. Two of these methods are descibed in chapter 3.

2.2.2 Dynamic Scheduling Methods

The following DLS methods are used in this thesis. Each method is explained and a chunk-size calculation formula is given. Some variables are common among the methods and are descibed in Table 2.1.

2.2.2.1 Self Scheduling (SS)

SS [2] describes a scheduling method where iterations are assigned to PUs during runtime of an application one at a time. SS is highly dependent on scheduling overhead time. If threads rely on a master thread to distribute iterations, the scheduling overhead time will become substantial. To reduce this time, the PUs should pull iterations from a common pool. The chunk size is always equal to one. SS is the most basic DLS and tends to balance the workload in the best possible way. It was first proposed for usage on applications where the single iteration execution times are not known and are expected to vary heavily.

2.2.2.2 Fixed Size Chunking (FSC)

Fixed Size Chunking (FSC) [3] describes a method, which calculates the optimal chunk size for self scheduling. It needs an input of the standard deviation σ and the overhead time h. Assumptions are made, that the overhead time is independent of the amount of iterations scheduled at once. A single chunk size is calculated and iterations are scheduled to each PU according to that chunk size whenever the PU idles. For this method, the mean iteration time μ, standard deviation of iteration times σ and the overhead time h must be known beforehand. The FSC method was first proposed as a way to calculate chunk sizes for dynamic scheduling to reduce the scheduling overhead time of SS while retaining a good load balance. The formula to calculate the FSC chunk size is written as in Equation 2.1.

$$Cs = \frac{\sqrt{2}Nh}{\sigma P\sqrt{log(P)}^{2/3}} \tag{2.1}$$

where Cs denotes the chunk size, N the amount of iterations and P the amount of available PUs.

2.2.2.3 Guided Self Scheduling (GSS)

GSS [4] distributes decreasing chunk sizes across the PUs according to Equation 2.2:

$$Cs = \frac{R}{P} \tag{2.2}$$

If R falls below P, the chunk size is set to 1. GSS was proposed as the first DLS method to be independent of μ, σ and h while scheduling chunks of iterations with a reduced scheduling overhead time over SS.

2.2.2.4 Trapezoid Self Scheduling (TSS)

Trapezoid Self Scheduling (TSS) [5] takes two inputs from the user. A starting size and an end size. The first chunk will be assigned according to the starting size, the last chunk according to the end size. The method calculates a linear decrease in chunk sizes for the iterations in between. A general suggestion for the input size is

$$\frac{N}{2*P}. \tag{2.3}$$

The chunk size is then calculated according to Equation 2.4. The variable t is the number of the current scheduling operation.

$$\begin{aligned} f &= start\_size, \\ l &= end\_size, \\ big\_N &= \tfrac{2I}{f+l}, \\ \delta &= \tfrac{f-l}{big\_N-1}, \\ Cs(1) &= f, \\ Cs(t) &= Cs(t-1) - \delta. \end{aligned} \tag{2.4}$$

2.2.2.5 Factoring (FAC)

Factoring (FAC) [6] is a batched scheduling method. In FAC chunks sizes are calculated for batches of iterations. Each batch contains the number of iterations calculated in Equation 2.5 or Equation 2.6 times the number of cores P. FAC is a mixture of FSC and GSS. In GSS, each batch only contains one chunk and FSC results, if there is just one single batch calculated with Equation 2.5. The j in the following equations is the batch-number.

$$\begin{aligned} R_0 &= N, \\ R_j + 1 &= R_j - PF_j, \\ Cs_j &= \tfrac{R_j}{x_j P}, \\ b_j &= \tfrac{P\sigma}{2\sqrt{R_j}\mu}, \\ x_0 &= 1 + b_0^2 + b_0\sqrt{b_0^2+2}, \\ x_j &= 2 + b_j^2 + b_j\sqrt{b_j^2+4}, \quad j > 0 \end{aligned} \tag{2.5}$$

In its original form, FAC needs an input of μ and σ. A simplified version is developed in the same paper with x set to 2 [6]. Its calculation formula is shown in Equation 2.6.

$$\begin{aligned} Cs &= \tfrac{R}{2P} \\ &= (1 - \tfrac{1}{2})^j \tfrac{N}{2P} \\ &= (\tfrac{1}{2})^{j+1} \tfrac{N}{P} \end{aligned} \tag{2.6}$$

Compared to the other DLS methods, factoring generates fairly little scheduling overhead while also being resistant to heavily varying iteration times.

2.2.2.6 Weighted Factoring (WF)

Weighted Factoring (WF) [7] works similar to normal factoring. After calculating the batch- and chunk size, the result is multiplied by a weight, which represents the speed of a PU

relative to the others. All the weights should add up to the number of PU. The chunk size calculation shown here incorporates $Cs_{factoring}$, which can be found in Equation 2.5 or Equation 2.6.

$$Cs = w_P * Cs_{factoring}. \tag{2.7}$$

It is the only nonadaptive DLS in this thesis which addresses varying PU speeds.

2.2.2.7 Taper Strategy (TAPER)

The Taper Strategy (TAPER) [8] is based on GSS. The difference is, that it takes the mean of the execution times of all iterations μ and their standard deviation σ into account to get a better load balance. On top of that a variable called α is used. This variable is influenced by the overhead time and the ratio of N to P.

$$\begin{aligned} T &= \tfrac{R}{P}, \\ v_\alpha &= \tfrac{\alpha\sigma}{\mu}, \\ K &= max(K_{min}, (T + \tfrac{v_\alpha^2}{2} - v_\alpha\sqrt{2T + \tfrac{v_\alpha^2}{4}})) \end{aligned} \tag{2.8}$$

If $\sigma = 0$ TAPER will yield the same chunk size as GSS. TAPER tries to achieve optimal load balance, while scheduling the largest possible chunk size Cs.

2.2.3 Adaptive Scheduling Methods

Adaptive Scheduling gets its name from its nature to adapt to its environment. May that be different PU speeds, interference, or an irregular input. During the runtime, mean execution times, standard deviation and overhead time of each PU are measured and used for calculating the next chunk sizes. By constantly adapting, these methods stay very close to the optimal line between perfect load balance and least amount of scheduling operations.

2.2.3.1 Bold Strategy (BOLD)

The Bold Strategy (BOLD) [9] is, as its name suggests, described as a bolder version of FAC. It takes multiple inputs and then generates additional values during the runtime for an adaptive chunk size calculation. Just like the TAPER strategy, it uses both mean execution time and standard deviation. An estimate of the overhead time of this method is also used.

A few additional important variables are: *boldm*, *boldn* and *totalspeed*. *Boldm* is "the number of iterations that are either unassigned or belong to chunks currently under execution." [9] Any time a PU finishes the execution of a chunk, *boldm* is decreased by that amount.

Three timers are used in the calculation of the next variable, *boldn*. t_1 is the time, when a PU starts calculating a new chunk. t_2 is the time, when a PU finishes its work and t is the last time before t_2 when any other PU finished the execution of its workload.

Boldn is described as "an estimate of the number of iterations that have not yet been executed." [9] At time t_2, *boldn* is decreased by $(t_2 - t)totalspeed + Cs - (t_2 - t_1)\frac{Cs}{Cs\mu+h}$.
"The value of *totalspeed* indicates the expected number of iterations completed per time unit, taking the allocation delay into account, and tends to lie slightly below $\frac{P}{\mu}$." [9] Each time a PU starts working on a chunk of size Cs, it increases *totalspeed* by $\frac{Cs}{Cs\mu+h}$. After the execution of this chunk, it decreases *totalspeed* by the same amount.
Taking all these values into account, the values of BOLD are initialized as in algorithm 1 and the chunk size is calculated according to algorithm 2.

Algorithm 1 BOLD initialization [9]

$a = 2(\frac{\sigma}{\mu})^2$
$b = 8a \ln(8a)$
if $b > 0$ **then**
    $ln\_b = \ln(b)$
end if
$p\_inv = \frac{1}{P}$
$c_1 = \frac{h}{\mu \ln(2)}$
$c_2 = \sqrt{2\pi c_1}$
$c_3 = \ln(c_2)$
$boldm = N$
$boldn = N$
$totalspeed = 0$

Algorithm 2 BOLD chunk size calculation [9]

$Q = \frac{R}{P}$
if $(Q \leq 1)$ **then**
    $Cs = 1$
else
    $r = max(R\ ,\ boldn)$
    $t = p\_inv * r$
    $ln\_q = \ln(q)$
    $v = \frac{Q}{b+Q}$
    $d = \frac{R}{1+\frac{1}{ln\_Q}-v}$
    if $(d \leq c_2)$ **then**
        $Cs = t$
    else
        $s = a(\ln(d) - c_3)(1 + (\frac{boldm}{rP}))$
        if $(boldb > 0)$ **then**
            $w = log(v * ln\_q) + ln\_b$
        else
            $w = log(ln\_q)$
        end if
        $q = min(t + max(0\ ,\ c_1 w) + \frac{s}{2} - \sqrt{s(t + \frac{s}{4})}\ ,\ t)$
    end if
end if

2.3 OpenMP

The OpenMP specification has implementations in many compilers, ranging from well-known compilers such as GNU Compiler Collection (GCC) and Intel XL, to small scale ones such as Mercurium. All compilers use either C/C++ or Fortran source code. OpenMP can be used for general parallelization, creation of threads, and work sharing, among other things. The focus in this thesis lies on the work sharing aspect of OpenMP. More specifically on the use of the omp for and omp do for the parallelization of loops. OpenMP is used inside the code by specifying compiler directives, for example #pragma omp parallel for to parallelize a for loop in C. Inside the OpenMP pragmas, many clauses can be set. For example, for making variables private to each thread, for synchronization of all threads, or for schedule clauses. OpenMP has the advantage over other programming paradigms, that it is not platform specific and is, therefore, portable. The developers do not need to concern themselves with the details. Only simple pragmas must be specified. The loop scheduling or the work distribution for example are handled by the OpenMP implementation. OpenMP adapts automatically to the specified or available amount of PUs and can even run the same code in sequential fashion.
On the other hand, OpenMP only works with shared-memory environments and the compiler needs to have an OpenMP implementation, like libgomp for GCC [1].

3
Related Work

All of the DLS algorithms in this thesis have been implemented into applications systems before, but only few DLS are specified in the OpenMP specification. Additionally other static loop scheduling algorithms have been implemented into some implementations of the OpenMP specification. Closest related to this thesis is the work of Penna et al. on loop scheduling in OpenMP. They proposed two new workload-aware loop scheduling algorithms and implemented them into libgomp as a proof of concept. The algorithms are called Smart Rount-Robin (SRR) [10][11] and BinLPT [12]. As the code of these loop scheduling algorithms is publicly available, their version of libgomp was used as a base for the implementation of the DLS in this thesis. While we used the same base of implementation, the loop scheduling methods considered herein differ greatly from SRR and BinLPT. Their focus lies in preprocessing the length of each iteration and then evenly distributing the iterations onto available threads, creating an almost perfect load balance before the actual scheduling occurs. The methods of Pemma et al. are described as being static, but their implementation inside of libgomp is dynamic. The implementation of SRR inside of libgomp looks similar to a dynamic algorithm in the sense, that every iteration is assigned separately according to a preprocessed task map. BinLPT is implemented differently than SRR in that it assigns chunks of loop iterations at a time and not single iterations. One other significant difference to our implementation of DLS methods is, that threads in the methods of Penna et al. do not loop endlessly, as described in chapter 5, until they find a free chunk of iterations, since every iteration is assigned to a specific thread beforehand.

A more dynamic loop scheduling algorithm was implemented by Durand et al. [13]. They introduced a new OpenMP loop scheduler called Adaptive. As the name suggest it adapts the chunk size of loop iterations to the application at hand via work stealing. It starts with the same distribution of iterations among threads as the static scheduler. Each thread obtains $\frac{N}{P}$ iterations. When one thread becomes idle, it steals half of the remaining workload from a loaded thread.

Every extension to the OpenMP scheduler mentioned in this thesis has been done in the schedule clause of libgomp. The SRR, BinLPT and all of our DLS are accessible from the runtime schedule by using environment variables. The Adaptive schedule can be used directly in the schedule clause.

Penna et al. designed their schedulers to be workload aware, which none of the other algorithms are. The Adaptive scheduler is the first one to use work stealing in OpenMP. Each of these three methods assigns a certain part of the workload to each PU beforehand, while our methods are nondeterministic and assign iterations dynamically during the runtime to available PUs. This way, between different executions of the same experiment, each PU might be assigned different iterations.
While Durand et al. and this thesis used known methods to implement new schedulers into OpenMP, Penna et al. employed a new way of creating scheduling strategies.

4

Dynamic Loop Scheduling in OpenMP

If an algorithm exhibits parallelism, parts of it can be run in parallel on multiple PUs. This parallelism can then be exposed inside of a scientific application using this algorithm. OpenMP is one of the ways to express this parallelism to the compiler, to give clear instructions which commands are going to be executed in parallel on which PUs. The full ability of the available PUs can be exploited this way to speed up the application run time. In this thesis we extended the expression of parallelism in OpenMP. With the newly implemented DLS methods, the underlying hardware, the PUs, can be exploited better than before for many irregular applications. Every existing DLS has its niche of applications and systems where its usage leads to the best performance.
Each loop scheduling method in OpenMP differs in its achieved load balance and the overhead and scheduling time. Static scheduling requires the least amount of overhead and scheduling time but leads to poor load balance on irregular applications. SS on the other hand can achieve almost perfect load balance, but its overhead time suffers greatly from N different scheduling operations. GSS and the newly implemented DLS try to find a balance between the two extremes SS and static scheduling. They reduce the amount of needed scheduling operations over SS but achieve slightly less perfect load balance.
By only modifying the GCC runtime library, the newly implemented methods can be used by other developers without the need to recompile the whole compiler. Only the schedule clause inside their applications needs to be changed. This way a wide range of new algorithms is made available to everybody parallelizing with OpenMP.

4.1 Area of application of dynamic loop scheduling algorithms

From working with the different benchmark suites and the DLS, we can recommend their usage as follows. The static scheduling should be used for any regular applications. There should be no big difference in single iteration execution time. Another usage for static scheduling is, if N is close to P. Any other method used in this case, would only introduce additional overhead. SS is best used for highly irregular applications, where no previous information about iteration length is known. For problems with a big N, SS is not recommended, since the amount of scheduling operations is also N. For such problems GSS is

better suited. The application area and performance of FAC is close to GSS. It is supposed to deliver a better load balance than GSS but should only be used, if the PU speeds do not differ between PUs. With differing PU speeds, WF should be used. TSS should be used, if a good start and end value are known or can be found out easily. Its performance can beat both GSS and FAC in their respective application area. TAPER excels at highly irregular applications, when the values for σ, α and μ are known. TAPER trumps BOLD in a reduced overhead time, which means it's better used for applications with a lot of iterations and smaller iteration length. BOLD should only be used if single iteration lengths are reasonably large, to allow for an accurate measurement of compute time. FSC leads to excellent execution times, if σ and h are known. The overhead time is a lot smaller compared to the other implemented DLS methods with input values in libgomp, since no calculations have to be made after the initial calculation of the chunk size. Its overhead time is very similar to that of SS, but it usually requires less scheduling operations.

4.2 Loop scheduling with libgomp

The loop scheduling in libgomp is not based on a master-slave architecture, but instead all threads are working on the iterations. At the start of a parallel section, a master thread is declared, but its function is only to initialize the parallel region (or loop in our case). Afterwards it behaves just like the other threads. After the initialization each thread executes the same function according to the loop schedule chosen. In this function a thread tries to allocate a certain amount of loop iterations (ranging from 1 to all of the available iterations, depending on the scheduling algorithm). If successful it means no other thread chose these loop iterations and it can go ahead and execute them. If another thread already allocated the first chosen iteration, the thread has to start again, trying to allocate iterations from the current latest iteration. Since a thread only ever allocates concurrent iterations, this will not lead to any iterations being left out. libgomp currently has 3 scheduling methods implemented. They are called *static*, *dynamic* and *guided* and are close, but not exact implementations of static scheduling, SS and GSS. In libgomp the chunk size can be specified for all existing methods. For *static*, this means that the scheduled iterations for each PU are not concurrent, but rather in blocks with size according to the specified chunk size.
Dynamic dynamically schedules blocks with size according to the specified chunk size during runtime of the application and distributes them whenever a PU idles.
The chunk size parameter is used as a minimum chunk size for the *guided* method. Additionally its chunk size calculation differs slightly from Equation 2.2 as it adds $P-1$ to R in the denominator.

4.3 Changes within libgomp

The runtime library libgomp on its own can be compiled and used as a shared library during runtime of an application. This means that changes can be made to it without the need to recompile either GCC or the application. Figure 4.1 gives an insight into the structure of libgomp. There is a function inside of GCC, which gets called whenever a #pragma

omp parallel for is encountered in an application. This function calls the initialization functions GOMP_loop_runtime_start and initialize_env inside of *loop.c*, blue border, and *env.c*, green border, if the clause schedule(runtime) is set. The function gomp_loop_init contains the initialization of variables used for the scheduling methods. It needed to be extended to implement new DLS. The functions inside of *env.c* are used to read in the schedule(runtime) from the environment variable OMP_SCHEDULE. This input is then used inside of *loop.c* to choose the functions matching the specified scheduling method. In the example of Figure 4.1, the FAC method is chosen. When the initializations are done, a function of *iter.c*, red border, is called. *iter.c* had to be extended by one method for each new DLS. The chunk size is calculated inside of *iter.c* and the PUs start working on their chunks from here.

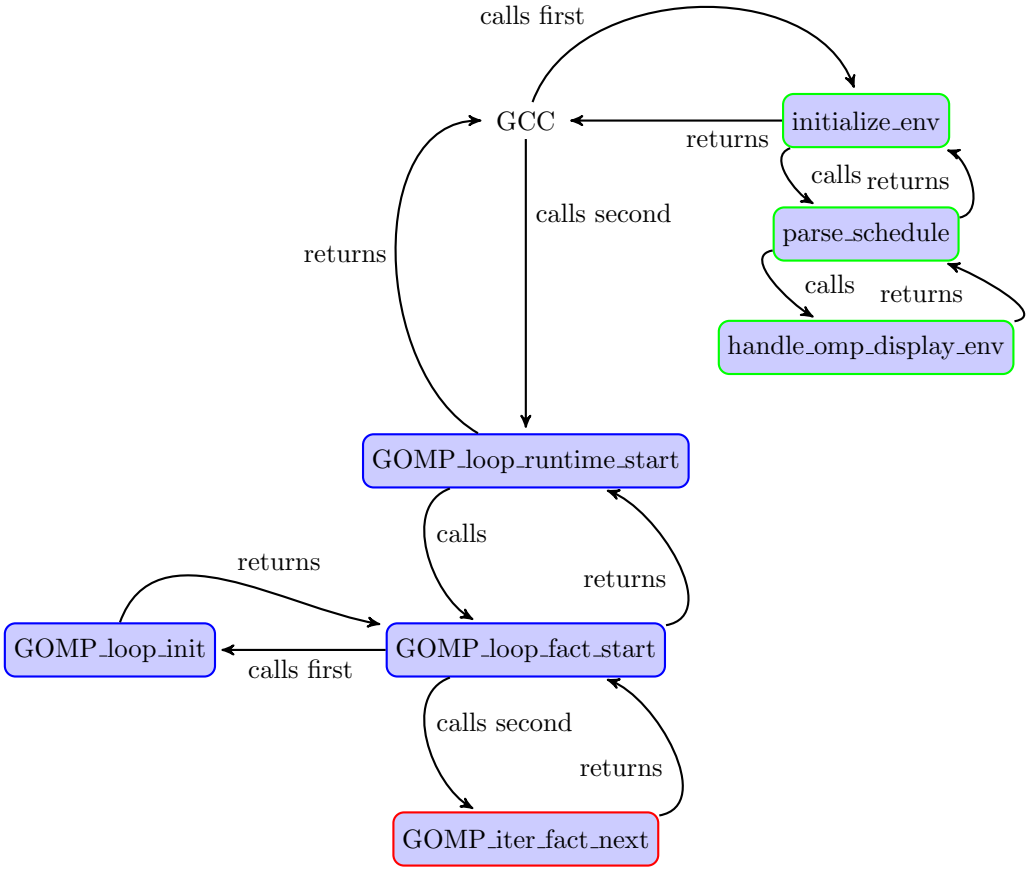

Figure 4.1: The modified libgomp routine call stack. The functions of *env.c* are outlined in green, *loop.c* is blue and *iter.c* is marked red.

4.4 Parallelization with libgomp on Linux

In Figure 4.2 the workflow of parallelizing a C program is shown. First the program code needs to be modified to include #pragma omp on top of the parallel region. If a loop is parallelized, #pragma omp parallel for schedule(runtime) should be used. In this example,

the scheduling method will be read in from the environment variable OMP_SCHEDULE. After the program has been modified accordingly, it can be compiled with $gcc - c - fopenmp$ to create object files. These object files are then linked with $gcc - o$ into an executable. The last two steps can be combined in one step, if no explicit linking is required. Only after the executable has been created, libgomp gets actually used. During the runtime of the executable, function calls to OpenMP are redirected to libgomp, if it is specified as the runtime library by setting the environment variable LD_LIBRARY_PATH to the path containing the libgomp runtime library.

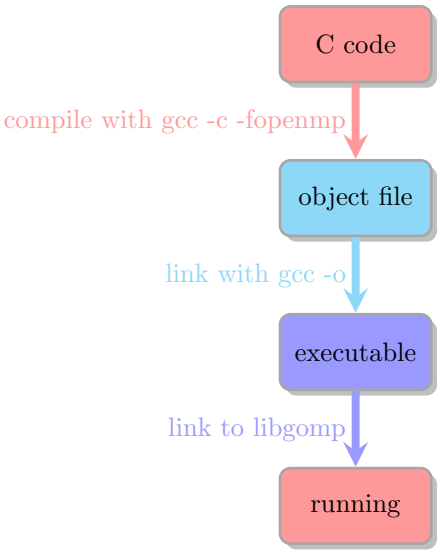

Figure 4.2: High level workflow of creating a parallelized program with OpenMP.

4.5 Implementation Decisions and Limitations

A decision was made to use the GCC implementation of OpenMP. The other considered implementation was the Intel OpenMP runtime library. Both of these implementations are open source and publicly available, but only the GCC is open source. The Intel compiler is not, which is the main reason why the GCC implementation libgomp was chosen. Furthermore in a comparison of the source code of libgomp and the Intel OpenMP runtime library, the libgomp code was found to be more extensible due to its simple and uncomplicated nature. In libgomp only few functions needed to be extended. In the Intel OpenMP runtime library, many templates would have needed to be modified to implement new DLS methods. During the implementation of adaptive DLS, the limitations of libgomp were revealed. Adaptive Factoring relies on a time measurement of single iterations. There is only one function, which gets called whenever a chunk needs to be calculated in libgomp. The only way to call this function for every single iteration, would be to set the chunk size to one. This would only lead to a version of SS with more overhead.

5 Implementation

This section specifies the extensions done in libgomp. Its purpose is to clarify the process of extending libgomp to help understand the effect of including new scheduling methods.

Algorithm 3 Loop initialization

```
1: procedure GOMP_LOOP_INIT
2:     switch schedule do
3:         case dynamic
4:             initializations
5:         case new
6:             read environment variables
7:             precompute constants
8: end procedure
```

Algorithm 4 Calculate next chunk

```
1: procedure GOMP_ITER_SCHEDULE_NEXT
2:     while 1 do
3:         get current starting point
4:         calculate chunk size
5:         end = start + chunksize
6:         if current start is still free then
7:             break
8:         end if
9:     end while
10:    newstart = end
11:    start executing iterations
12: end procedure
```

The loop scheduling clause in OpenMP currently has 4 options: *static*, *dynamic*, *guided* and *runtime*. SS is called *dynamic* in libgomp, GSS is called *guided*. A possibility to specify chunk sizes for these methods already exists in the libgomp specs by adding it into the schedule clause. The newly implemented DLS methods are made accessible through the OpenMP schedule clause runtime. By writing the desired scheduling method into the environment variable OMP_SCHEDULE, the chosen method will be used every time the schedule(runtime) is used in the code. While implementing the DLS methods, which are

covered in this thesis, the known implementation approaches used in the existing DLS methods of libgomp were followed. As a basis, the guided scheduling was used. There are a few different files inside the libgomp runtime, which needed to be edited to implement a new scheduling method, namely *env.c*, *libgomp.h*, *iter.c* and *loop.c*. The connection and function calls inside of these files can be found in Figure 4.1.
Inside the file *env.c* translation of the environment variable OMP_SCHEDULE is managed. This is used to tell the scheduler which scheduling method to use, when schedule(runtime) is specified by the programmer. In *libgomp.h* the methods of *iter.c* and *loop.c* are declared. In addition, the struct *workshare* is located here. Each parallel loop in OpenMP has one *workshare* assigned to it. Inside all the global variables for the new DLS methods are saved. The files *iter.c* and *loop.c* contain the loop scheduling algorithms themselves. In *loop.c* all variables used in the algorithms are initialized as shown in algorithm 3. The method containing the different initializations is called once for each #pragma omp parallel for and initializes every variable necessary for the chosen scheduling method. For example, the processor-weights are read from an environment variable for the weighted factoring schedule in *loop.c*. The file *iter.c* contains methods which get called every time a PU is idle inside a #pragma omp parallel for region. This means that every PU calls the function matching the chosen scheduling method at the start of the loop and then again every time it finishes its assigned workload. Generally the procedure looks like algorithm 4. In these methods the next chunk size is calculated according to the algorithms specified in the background section. After calculating the chunk size, a PU tries to allocate a chunk of work. If that chunk is already taken it tries again with a new starting point. The global starting point is updated every time a PU successfully allocates a chunk. Generally the PU start at iteration 0, but often times the first chunk allocated is not the first one to complete. Since all the implemented scheduling methods are dynamic, there is no telling which PU will get which chunk of loop iterations. For the static methods the assignment is a lot more straightforward, since it is predetermined which PU gets assigned which chunk.
For the adaptive methods a timer is needed. The duration of each chunk-execution is measured and used in the calculation of the next chunk sizes. The method called after a chunk has been finished by a PU is again *iter.c* inside of libgomp, so the time measurements are happening at the start of this method and whenever a chunk is successfully allocated at the end of this method.
In the following sections multiple variables are used. The naming is according to Table 2.1. For every fraction a check is made to ensure the denominator is not equal zero. Every function in *iter.c* has 2 parts. An initialization part and an endless loop. The loop is restarted every time a PU fails to allocate a chunk. Only once a chunk is successfully allocated or when no more iterations are left, the loop can be exited. The chunk size calculation is always done inside of the loop. If the calculated chunk size is below one it is always set to one.

5.1 Fixed Size Chunking

For FSC [3] the environment variables SIGMA and FSCH are read in and used in the formula according to Equation 2.1: $\frac{\sqrt{2}Ih}{\sigma P\sqrt{\log P}}^{2/3}$ Furthermore, if the calculated chunk size is bigger than N, it is set to N. If it is smaller than one, it is set to one.
As in all the other methods, in the loop of *iter.c* the chunk size is used to try to allocate a chunk.

5.2 Factoring

Of all the implemented methods, FAC [6] has the smallest initialization overhead. Inside of *loop.c* only two variables are initialized, the maximum workload and a counter. The workload is used in the chunk size formula according to chapter chapter 2. In the initialization part of its *iter.c* function each PU increments the global counter and saves its assigned number. In the loop, this number is used to calculate which batch is currently being assigned with $batch\_number = \frac{counter}{P} + 1$.
The chunk size calculation is done according to $Cs = \frac{N}{2^{batch\_number} * P}$, which is equivalent to the last line in Equation 2.6.

5.3 Trapezoid Self Scheduling

In TSS [5] the initialization method in *loop.c* reads the environment variables TRAPSTART and TRAPEND. From these, according to Equation 2.4 a big_N and a δ are calculated with the formulas $big\_N = \frac{2*I}{start\_size+end\_size}$ and $\delta = \frac{start\_size-end\_size}{big\_N-1}$
Furthermore a counter is initialized, which is incremented in the initialization part of TSS' *iter.c* function. The chunk size is then calculated according to $Cs = start\_size-(\delta*counter)$.

5.4 Weighted Factoring

WF [7] uses the same formulas and algorithm FAC uses. The only difference is the addition of processor weights. These are read from the environment variable WEIGHTS inside the initialization function of *loop.c*. Further initializations are done for the maximum workload and the counter. In *iter.c* the weights are then multiplied with the same formula as FAC: $Cs = w_i * \frac{N}{2^{batch\_number} * P}$.

5.5 Taper Strategy

The TAPER [8] strategy is the first method to read in a μ and σ. These are read from the environment variables MEAN and SIGMA respectively. Another variable, α is read from ALPHA. Then the global variable v_α is calculated with $v_\alpha = \frac{\alpha\sigma}{\mu}$ from Equation 2.8
This global variable is then used inside the loop of TAPERs *iter.c* function together with the remaining workload T_i to calculate the chunk size according to
the formula $Cs = T_i + \frac{v_\alpha^2}{2} - v_\alpha\sqrt{2T_i + \frac{v_\alpha^2}{4}}$.

5.6 Bold Strategy

As in the method above, for BOLD [9] μ and σ are also read in from environment variables. For the μ and σ, the same ones are used as for TAPER. Additionally the overhead time h is read in from BOLDH. From these variables and P and N the method calculates the constants according to algorithm 1. Then the global variables *boldm*, *boldn totalspeed*, *boldtime* and three arrays, *boldarray*, *speedarray* and *timearray* are initialized to 0 and the current time respectively. *boldarray* will hold the values of every Cs and *speedarray* the values of each *totalspeed* calculated separately for every PU.

At the start of the function in *iter.c*, each PU initializes the local variable t to the current value of *boldtime*. Afterwards *boldtime* is updated with the current time. A second local variable, t_2 is initialized to the current time. Next the global variables *boldm* and *totalspeed* are altered by subtracting the values in *boldarray* at position P from *boldm* and in *speedarray* at position P from *totalspeed*. After this, the loop starts.

The chunk size is then calculated according to algorithm 2. If a PU successfully allocates a chunk, its size is saved in *boldarray*. Next the differences $t - t_2$ and $timearray[P] - t_2$ are calculated to be used in the calculation of *boldn*. The first difference is the difference between the last time a chunk has finished computing t and the starting time of this method on this PU t_2. The second difference is between the start of the last execution of the last chunk on this PU $timearray[P]$ and the end of it t_2. After *boldn* has been updated, *totalspeed* and $speedarray[P]$ are updated with the same number. At last t_2 is saved in $timearray[P]$ to be used for the next chunk size calculation on this PU.

6 Evaluation

The first part of this chapter contains explanations and specifications about the various benchmarks used for the experiments. Some general lines along which all the benchmarks were modified to use the new DLS methods are also given. In the second part of this chapter, the design, by which every experiment was performed, is explained. The second part contains the results of these experiments with explanations. The goal of this chapter is to visualize the advantages of the newly implemented DLS by comparing them to the existing OpenMP methods and to each other. Many experiments were done on the different benchmarks, but only the most interesting ones are presented in this chapter. The whole table of benchmarks used and all the experimental results can be found in Appendix A. Overall many of the experiments turned out to have regular parallel loops. While the newly implemented DLS methods outperform the existing DLS methods on these experiments, they can not reach the performance of static scheduling. Much more interesting are benchmarks with irregular parallel loops, where DLS can show their strength.

6.1 Benchmarks

The benchmark suites are written in C, C++ and Fortran. Only benchmarks, which already had an OpenMP implementation and at least one parallel loop were extended to use the new DLS. For most benchmarks a simple addition of schedule(runtime) to the #pragma omp for C/C++ code or to $omp do for Fortran was sufficient. Some of the benchmarks from the NAS Parallel Benchmark Suite (NAS) and from the SPEC OpenMP Benchmark Suite (SPEC OMP) contained multiple files with up to 93 different parallel loops. For these file, not every loop was parallelizable with DLS. Some of them require the scheduling to be done statically.

For the smaller benchmarks with less parallel loops, code was temporarily added to measure single iteration times. This was done to acquire means and standard deviations for the DLS requiring these as input. The measurements themselves were handled by the C library time. The function used was clock_gettime with CLOCK_MONOTONIC. Table 6.1, Table 6.2, Table 6.3 and Table 6.4 give a small explanation of each extended benchmark. More specific as to how they were extended, can be found in the next chapter.

A full list of experiments can be found in Table A.1.

Table 6.1: RODINIA Benchmarks [14]

Benchmark Suite	Benchmark Name	Explanation
Rodinia	b+tree	Uses binary tree search to find values matching a key.
	cfd	Computational Fluid Dynamics solver. Solves the three-dimensional Euler equations for compressible flow.
	hotspot	Iteratively solves a series of differential equations to estimate processor temperature.
	hotspot3D	HotSpot for three dimensional thermal models.
	kmeans	Clustering algorithm used for data mining. Tries to group observations into k different clusters.
	lavaMD	Calculates the potential of particles and their relocation due to mutual forces between particles within a large 3D space.
	leukocyte	Detects and tracks rolling white blood cells in videos of blood vessels.
	lud	A simple LU decomposition, which calculates the solutions of a set of linear equations.
	myocyte	Models a heart muscle cell and simulates its behavior.
	nn	Finds k nearest neighbors in an unstructured data set.
	nw	Global optimization method for DNA sequence alignment. (Needleman-Wunsch algorithm)
	particlefilter	Estimates the location of a target object from noisy measurements of that target's location and an idea of the object's path.
	srad	A diffusion algorithm based on partial differential equations. Used for removing the speckles in an image without sacrificing important image features (Speckle Reducing Anisotropic Diffusion)
	streamcluster	Finds a number of medians for a stream of input points, so that each point is assigned to its nearest center.

Table 6.2: OpenMP Source Code Repository (OMPSCR) Benchmarks [15]

Benchmark Suite	Benchmark Name	Explanation
OmpSCR	fft6	Computes the discrete Fourier transform of an input signal by using Bailey's 6-step Fast Fourier Transform algorithm.
	qsort	Sorts an integer array with the Quicksort algorithm.
	md	Calculates a simple molecular dynamics simulation, using the velocity Verlet algorithm.
	mandel	Estimates the Mandelbrot Set area using MonteCarlo sampling.
	fft	Computes the discrete Fourier Transform of an input signal.
	lu	LU reduction of a 2D dense matrix. Calculates the solutions of a set of linear equations.
	pi	Computes π.

Table 6.3: NAS Benchmarks [16] [17] [18]

Benchmark Suite	Benchmark Name	Explanation
NAS	ua	solution of a heat transfer problem in a cubic domain. Uses dynamic and irregular memory access.
	sp	Calculates a synthetic system of nonlinear partial differential equations using scalar pentadiagonal solver kernelsSolves a synthetic system of nonlinear partial differential equations using scalar pentadiagonal solver kernels.
	bt	Calculates a synthetic system of nonlinear partial differential equations using block tridiagonal solver kernels.
	lu	Calculates a synthetic system of nonlinear partial differential equations using symmetric successive over-relaxation solver kernels.
	mg	Approximates the solution to a three-dimensional discrete Poisson equation using the V-cycle multigrid method.
	ft	Calculates a three-dimensional partial differential equation using the fast Fourier transform.
	ep	Generates independent Gaussian random variates using the Marsaglia polar method.
	cg	Estimates the smallest eigenvalue of a symmetric positive-definite matrix using the inverse iteration with the conjugate gradient method as a subroutine for solving systems of linear equations.
	is	Sorts an integer array using bucket sort.

Table 6.4: SPEC OMP Benchmarks [19]

Benchmark Suite	Benchmark Name	Explanation
SPEC	wupwise	Wuppertal Wilson Fermion Solver. Solves the inhomogeneous lattice-Dirac equation with the BiCGStab iterative method
	swim	Solves a system of shallow water equations using finite difference approximations on a N1 x N2 grid.
	mgrid	Computes a three dimensional potential field with a multi-grid solver.
	equake	Simulates the propagation of elastic waves in large and highly heterogeneous valleys.
	apsi	Calculates temperature, wind, velocity and distribution of pollutants as prognosis.
	fma3d	Simulates the inelastic, transient dynamic response of three-dimensional solids and structures subjected to impulsively or suddenly applied loads.
	art	Adaptive Resonance Theory. Recognizes objects in an image with a neural network.
	ammp	Solves differential equations to run a molecular dynamics simulation.

Table 6.5: Design of experiments. Each experiment was executed with 20 threads on the miniHPC system.

Benchmark suite	Kernels	Problem sizes	input commands	input files	Total # of loops parallelized with DLS	# of runs	DLS methods
RODINIA	b+tree		j 6000 3000 k 10000	mil.txt, command.txt	1	100	STATIC, FSC, GSS, TSS, FAC, WF, TAPER, BOLD, SS
	cfd			missile.domn.0.2M	5		
	hotspot	1024x1024, 2 iterations	1024 1024 2 20	temp_1024, power_1024	1		
	hotspot3D	264x264, 20 layers, 100 iterations	264 20 100	power_512x8, temp_512x8	1		
	kmeans			kdd_cup	1		
	lavaMD		boxes1d: 10		1		
	leukocyte	5 frames		testfile.avi	3		
	lud			2048.dat	2		
	myocyte	end of sim interval: 1000, instances of sim:1000	1000 1000 1 20		1		
	nn	k: 20, lat: 60, long: 100	20 60 100	filelist_4	1		
	nw	8192x8192, penalty: 1000	8192 1000 20		2		
	particlefilter	128x128, 100 frames, 20000 particles	-x 128 -y 128 -z 100 -np 20000		10		
	srad	1000 iterations, λ: 0.7 image: 502x458	1000 0.7 502 458 20		2		
	streamcluster	min: 10, max: 20, dimension: 256, 65536 datapoints, chunksize: 65536, clustersize: 1000	10 20 256 65536 65536 1000 none output.txt 20		2		
OmpSCR	fft6	4096, 50 iterations			4	100	
	qsort	2097152			1		
	md	65536, 50 iterations			2		
	mandel	524288			1		
	fft	65536			1		
	lu	5000			1		
	pi	1000000000			1		

Table 6.6: Design of experiments. Each experiment was executed with 20 threads on the miniHPC system.

Benchmark suite	Kernels	Input sizes	Total # of loops parallelized with DLS	# of runs	DLS methods
NAS	ua	S, W, A, B, C, D	77 in 11 files	1	STATIC, FSC, GSS, TSS, FAC, WF, TAPER, BOLD, SS
	sp	S, W, A, B, C, D	32 in 10 files		
	mg	S, W, A, B, C, D	13		
	lu	S, W, A, B, C, D	45 in 15 files		
	ft	S, W, A, B, C	8		
	ep	S, W, A, B, C, D	1		
	cg	S, W, A, B, C, D	18		
	bt	S, W, A, B, C, D	33 in 11 files		
	is	S, W, A, B, C, D	9		
SPEC	wupwise_m	refsize	16 in 10 files	1	
	swim_m	refsize	8		
	mgrid_m	refsize	12		
	equake_m	refsize	11		
	apsi_m	refsize	28		
	fma3d_m	refsize	93		
	art_m	refsize	1		
	ammp_m	refsize	7		

6.2 Design of Experiments

The OpenMP part of the Rodinia benchmark suite contains 19 different benchmarks. A number of these were used for experiments. The remaining benchmarks did either not compile, did not contain a parallel loop or did not output time measurements and therefore were omitted from the results. A small explanation to each Rodinia benchmark can be found in Table 6.1. The OMPSCR is divided into 15 different parts. Some of these 15 parts contain multiple benchmarks, but most of them only one. Seven of these 15 parts were used for experiments, the rest was omitted due to the same reason as some of the Rodinia benchmarks were omitted. The description of the OMPSCR can be found in Table 6.2. All of the ten different NAS benchmarks were used for an experiment. In this experiment, due to a execution time of multiple days, every benchmark was executed only once with every single DLS. The description of the different benchmarks can be found in Table 6.3. From the 11 different benchmarks of SPEC OMP, eight were used for experiments. The details to each benchmark can be found in Table 6.4

Some of the extended benchmarks contain multiple parallelized loops as seen in Table 6.7 through Table 6.10. These tables also contain information as to which loops were modified and which scheduling method was originally used. Specific lines are highlighted to show the most interesting benchmarks either due to the average iteration length, which is important for accurate μ and σ measurements or due to some loops containing dynamic scheduling in the original benchmark suite.

Table 6.7: Parallelized loops in each benchmark of the Rodinia benchmark suite. All of the scheduling clauses were changed from static to runtime.

Benchmark Name	Total # of parallel loops	average iteration length in seconds
b+tree	1	0.000001
cfd	5	less than 0.0000001
hotspot	1	0.00011
hotspot3D	1	0.0004
kmeans	1	0.3
lavaMD	1	0.001
leukocyte	3	find_ellipse: 0.0042, track_ellipse: 0.15
lud	2	0.000024, 0.000002
myocyte	1	0.013
nn	1	0.000008
nw	2	0.000004, 0.000002
particlefilter	10	1-3: 0.0000001, 4: 0.000001, 5-9: 0.00000001, 10: 0.000004
srad	2	0.00002, less than 0.0000001
streamcluster	2	0.000001, less than 0.0000001

For every parallelized loop inside OMPSCR and the Rodinia benchmarks, measurements were taken to find the right input values for the DLS requiring them. The overhead times were measured to be 0.000005 seconds for BOLD and 0.0000005 seconds for FSC and used for every benchmark. For NAS and SPEC OMP, a general 0.0005 seconds was used as the μ and 0.000005 seconds was used as input for the σ. The TSS starting point was set to the recommended $\frac{N}{2*P}$ and the endpoint usually set to 1. In cases with a big N, the endpoint

Table 6.8: Parallelized loops in each benchmark of the OMPSCR. All of the scheduling clauses were changed from static to runtime.

Benchmark Name	Total # of parallel loops	average iteration length
fft6	4	1-3: 0.000005 , 4: 6.334337
qsort	1	0.00005
md	2	0.00389, less than 0.0000001
mandel	1	0.000218
fft	1	0.00000001
lu	1	0.000065
pi	1	0.00000001

Table 6.9: Details of parallelized loops in each benchmark of NAS.

Benchmark Name	Total # of parallel loops	# of parallel loops modified	Original scheduling
UA	77 in 11 files	all	static
SP	32 in 10 files	1-12, 18-23, 26-32	static
MG	13	all	static
LU	45 in 15 files	1-4, 7-11, 14, 16-28, 30-35, 38, 42, 45	static
FT	8	all	static
EP	1	all	static
CG	18	all	static
BT	33 in 11 files	1-10, 18-24, 27-33	static
IS	9	1-4, 7-9	8 times static, 1 dynamic

Table 6.10: Details of parallelized loops in each benchmark of SPEC OMP.

Benchmark Name	Total # of parallel loops	Total # of parallel loops modified	Original scheduling
wupwise	16 in 10 files	all	static
swim	8	all	static
mgrid	12	all	static
equake	11	all	static
apsi	28	all	static
fma3d	93	all	static
art	5	only the 5th	4 times static, 1 time dynamic
ammp	7	all	5 times static, 2 times guided

was set to a respectively higher value. The weights in WF were set to 1 for every PU, since the setup did not allow for different processing speeds. For the experiments with TAPER, α was set to 1.3 according to the work of Steven Lucco [8].

Every experiment was run on the same hardware as specified in Table 6.11 with the same implementation of libgomp. The benchmarks themselves were compiled with GCC version 5.4.0. GCC and libgomp were used for benchmarks written in C and C++ as well as benchmarks written in Fortran.

Each experiment consists of a whole benchmark suite run sequentially on one node of the cluster. The information about one node can be found in Table 6.11. A very broad range of

Table 6.11: Characteristics of a single miniHPC cluster node

Processor Name	Intel(R) Xeon(R) CPU E5-2640 v4
Processor Speed	2.40GHz
Processor Architecture	x86_64
L1d cache	32k
L1i cache	32k
L2 cache	256k
L3 cache	25600k
physical CPUs	20
virtual CPUs	40
RAM	64GB

benchmarks was used, some even twice in different benchmark suites to cover all cases and find patterns as to which schedule type fits to which kind of problem.
Each experiment was done with the number of OpenMP threads set to 20. Thread pinning (via GOMP_CPU_AFFINITY) improved the results by a few percent and reduced the variance in between runs. The overhead of the different schedule types was estimated by a measurement inside of the libgomp runtime. This overhead time was then used as an input to the BOLD and FSC methods.
Each used benchmark outputs an execution time. This time was used as the primary measurement, by which the DLS were compared.
In addition to the modified benchmarks, each benchmark suite was run in its original form once. The results of these experiments did not vary much from the usage of static scheduling normally and dynamic scheduling for *art* and *IS*. They are included in the figures as red arrows on the y axis. For the NAS figures, the shape of the respective size was used instead of arrows.
To show how load balance in a highly irregular application looks like, Score-P was used with tracing on a LU decomposition. The execution with static scheduling is highly inefficient. The loop execution time can be seen in Figure 6.1 in brown, while the purple parts represent the time a thread was idle. The relation of idle to busy time of threads on the static execution (the lower part of the picture) is 1:1, which almost doubles the total time taken.
In the top half of Figure 6.1 an almost perfect load balance can be seen. There are no purple parts to be seen. This means every thread is busy from start to finish and the overall time is only composed of time spent working.

6.3 Results

Looking at the experiments overall, the static method achieved best results in most of the experiments. This leads to the conclusion, that most of the benchmarks were more regular than irregular. While this is the case, small Ns can also lead to a good performance of static scheduling over the other scheduling types. The DLS methods FAC and WF are supposed to offer similar results, since their chunk size calculations are the same. FAC has the advantage, that it does not rely on an input and still offers good results. It outperforms static scheduling on irregular problems and SS on regular problems. It never outperforms every other method, but reliably offers good results. The problems of the SS method with

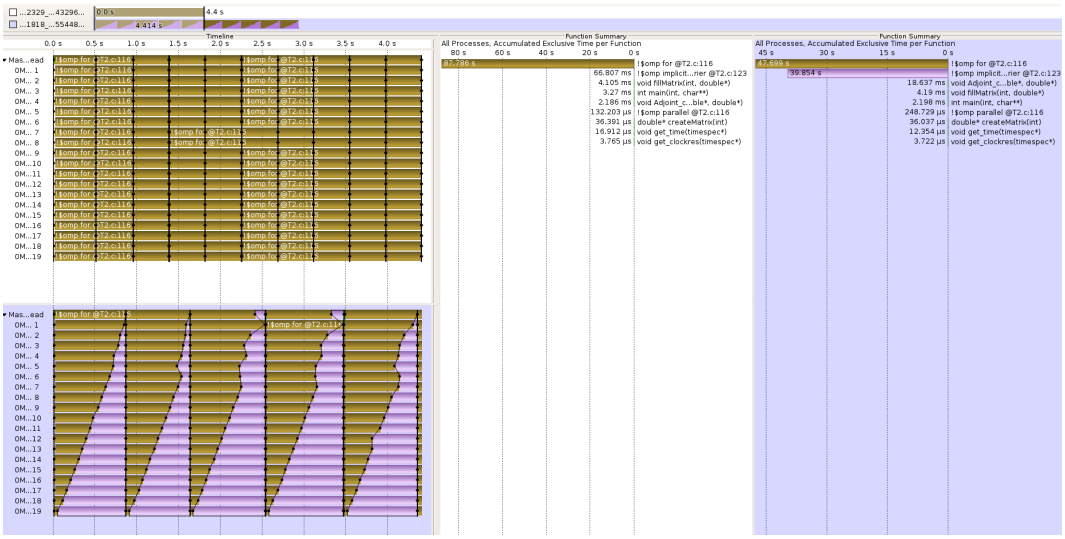

Figure 6.1: Comparison between static and dynamic scheduling load balance of 20 threads working on a LU decomposition 10 times.

regular load on loops with a high number of iterations can be seen on many results. TSS, TAPER and FSC face similar problems, as they need good input values to produce good results. On some of the benchmarks these values were chosen badly, which reflects in their respective performance. The BOLD method requires on top of the input values, to have a measurable iteration length. Small iteration times lead to big uncertainties in the time measurements and therefore bad performance.

6.3.1 Rodinia benchmark suite

The Rodinia benchmark suite is the first one to be modified to include μ and σ for each loop. The same experiment has been run 100 times and the results have been summarized in Figure 6.2 and Figure 6.3. Shown are the median execution times as black dots, the standard deviation as black bar and the min and max values as small grey bar. The first interesting benchmark here is the cfd benchmark. It is regular and the N is high, leading to a bad performance by SS. The *hotspot* and *hotspot*3D benchmarks showcast a very good choice of TSS values and a very bad choice respectively. While *kmeans* has a very high iteration length, it is also rather regular. This means, the DLS did not do as well as the static scheduling. This could have been a good case for BOLD, but due to its regularity, did not quite work out. The $lavaMD$ and *leukocyte* benchmarks are the first irregular benchmarks. The performance of almost all DLS beats the performance of static except for TSS where the input values were chosen poorly. *Lud*, *nn* and *nw* seem all rather regular. They only really differ in the choice of input values. It is very unexpected to see static scheduling perform so well on a LU decomposition. The *myocyte* benchmark is another slightly irregular benchmark, resulting in similar performance from static scheduling and DLS alike. $Particlefilter$ is the most interesting benchmark of this suite. It contains irregularities, leading to a suboptimal performance of static scheduling. The 10 parallelized loops are all executed so frequently, that the SS method fails to achieve a reasonable performance. Due

to the amount of different loops and their differing and sometimes extremely short execution times, the methods requiring input values do not work well. This is one of the only benchmarks, where FAC, GSS are clearly better than the others. It seems to require a not too coarse but also not too fine grained chunk size and little overhead due to the amount of iterations. The three new methods with the least amount of overhead time reach the best timings here. *Streamcluster* performs similar to this. Even tho the input values for TAPER, BOLD and FSC are the same, the TAPER method completely fails. This can be amounted to the extremely small μ and not so small σ value, which leads to a badly computed chunk size of TAPERs. The *srad* benchmark is another regular benchmark with a medium amount of iterations. Badly chosen TSS input values lead to a bad performance. The one loop containing extremely small iterations leads to bad calculations of FSC chunk sizes. In comparison to the *streamcluster* benchmark, the μ and σ are both extremely small, resulting in a better calculation of the TAPER chunk size.

6.3.2 OpenMP SCR benchmark suite

Aligning with the Rodinia benchmark experiments, the OMPSCR benchmarks, as seen in Figure 6.4, were also run with specific μ and σ set for every parallel loop. In general there are more regular benchmarks in this suite and their execution times are higher, leading to a smaller dataset overall, since less experiments could be performed in the same time. $fft6$ is one of the only irregular benchmarks, leading to a slightly better performance of DLS compared to static scheduling. The sizes for TSS were chosen poorly. The fft benchmark is regular, has a lot of iterations and a small iteration time, leading to bad performance for any DLS with high overhead. The only DLS keeping up with static scheduling are GSS and FAC due to their small overhead time and comparably small number of scheduling operations. The *qsort* benchmark is another regular one. Most of the input values here were chosen poorly, which results in bad performance on any method requiring these, namely FSC, BOLD, TAPER and TSS. The *md* benchmark shows an extreme case of a very regular application with a huge amount of iterations. The bad performance of FSC can not be explained by a bad choice of input values, since TAPER and BOLD use the same σ. The *mandel* benchmark is highly irregular with a low amount of loop iterations. The loop iteration times are comparatively high and the BOLD method is therefore able to make good calculations and predictions, which leads to a good performance. The lu implementation of OMPSCR is slightly more irregular, but the static method still performs just as well as the best DLS methods. Lastly the *pi* benchmark is very regular with a huge amount of iterations. The SS method is the only one to perform badly here, showing well chosen input values for the other DLS methods.

6.3.3 NASA OpenMP parallel benchmark suite

For NAS, only one experiment was made. It took four days to complete and includes every benchmark run in every size on every scheduling method. The results do not carry much significance, since they stem from one single run only. Furthermore, every parallel loop was set to use the runtime schedule, which might not be optimal. In some cases there were

up to 77 different loops. Most smaller loops lose time on scheduling overhead due to this. The μ and σ values were set to the same number for every benchmark. Considering all of that, the UA benchmark has irregular and regular parts. The irregular parts overweight the regular ones, leading to an improvement in performance on using some DLS. For SS there are simply too many loop iterations overall, leading to too many scheduling operations and a bad execution time. The SP, MG, LU, FT and EP benchmarks are fairly regular and show no big difference in execution times between scheduling methods. This leads to the conclusion, that not many iterations are scheduled and the sequential part of these programs overweights the parallel part. The CG benchmark contains a slightly higher number of iterations, leading to a worse performance of SS. The other scheduling methods are still reasonably close in performance. The BT method shows signs of badly chosen input values on all methods requiring these. Lastly, the only definitely irregular application, IS shows good performance for every scheduling method, except for TSS due to badly chosen input values. The DLS generally beat static scheduling here, but not by a lot. Overall it looks like the NAS benchmarks contain a lot of sequential operations, leading to a very similar performance of all scheduling methods. This could be improved by changing the scheduling method of single loops and adapting input values. The graphs as seen in Figure 6.5 and Figure 6.6 are on a logarithmic scale, showing the extremely poor performance of SS on many benchmarks.

6.3.4 Spec OpenMP benchmark suite

Like the NASA OpenMP suite, SPEC OMP was also only executed once. One experiment was made with all benchmarks run for the size $refsize$ on each scheduling method. Once again, general input values for μ and σ were chosen. This lead to less comparable results overall and especially bad performance for FSC, TAPER and BOLD. Most of the input values for TSS were also once again chosen poorly.

The first benchmark, *wupwise*, contains multiple loops with a lot of iterations, leading to bad performance for SS. A badly chosen σ with no reference μ, lead to a very bad performance of FSC. The *swim* benchmark shows similar results to the *wupwise* benchmark. This time, the static method beat out the other methods by a big amount showing a very high regularity of the application. *mgrid* showcasts an abysmal choice of TSS input values, leading to a execution time multiple times higher than the other DLS. Once again, it is a regular application. *Equake* behaves like *wupwise* with another badly chosen σ and big number of iterations. *Apsi* and $fma3d$ are also regular benchmarks and behave very similar to the already mentioned benchmarks. *Art* and *Ammp* are the only irregular benchmarks in SPEC OMP. *Ammp* shows a general good performance of all new DLS methods, even tho the different input values were not specifically chosen for each loop. Therefore the more general method, FAC, scored the best result.

The benchmark *art* was specifically modified to only contain one parallel loop with schedule set to runtime. This loops single iteration execution times were measured and used for the calculation of μ and σ. The start size for TSS was set to follow the Equation 2.3 and the end size set to 1. It still performed considerably worse than the other, newer DLS. Although

all the input values were chosen carefully, the FAC and SS methods performed similar to TAPER, FSC and BOLD. More experiments should be run here to achieve more comparable results.

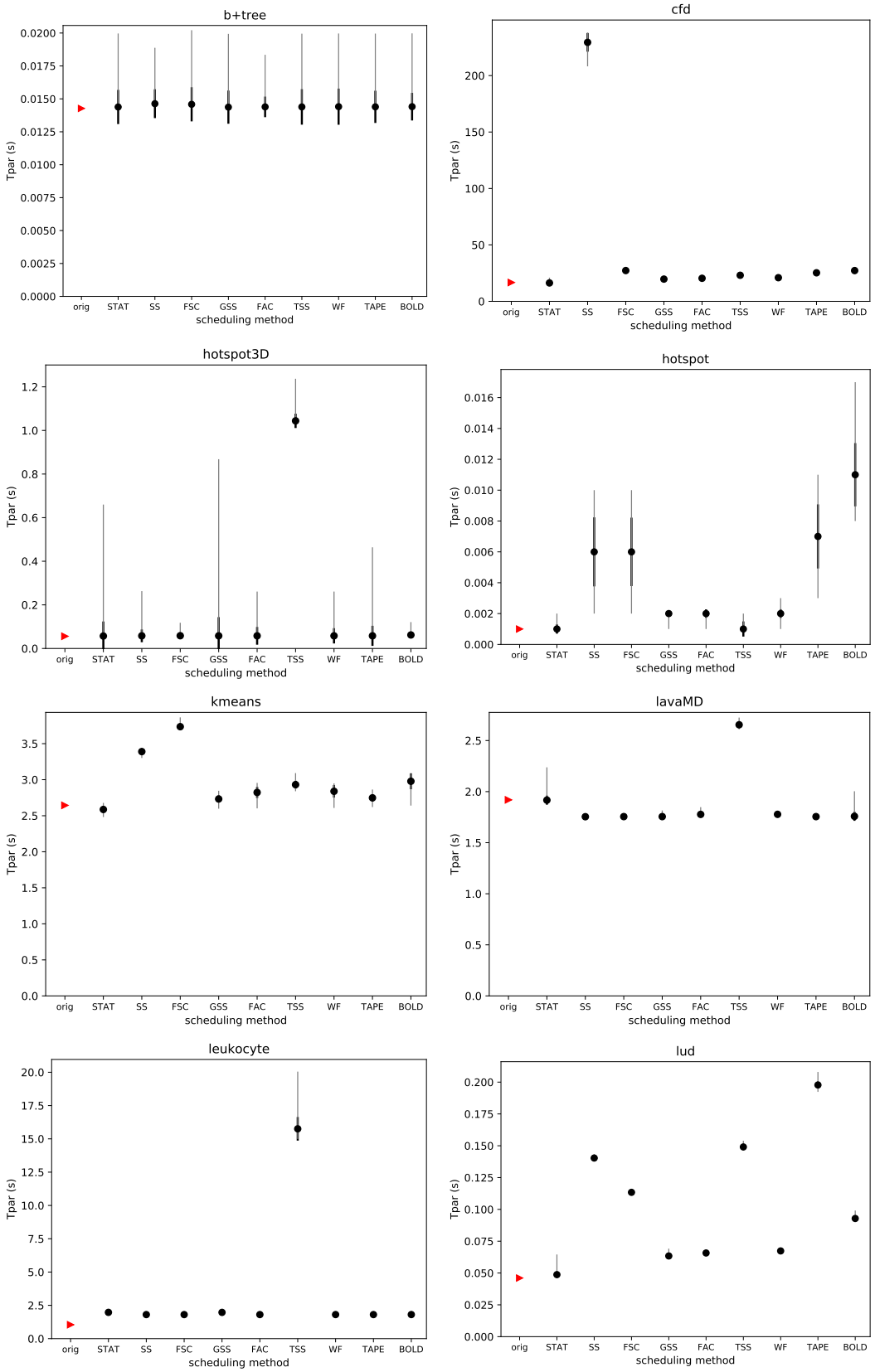

Figure 6.2: Medians, standard deviation, minimum and maximum values of the execution time of 100 runs of the Rodinia benchmark suite.

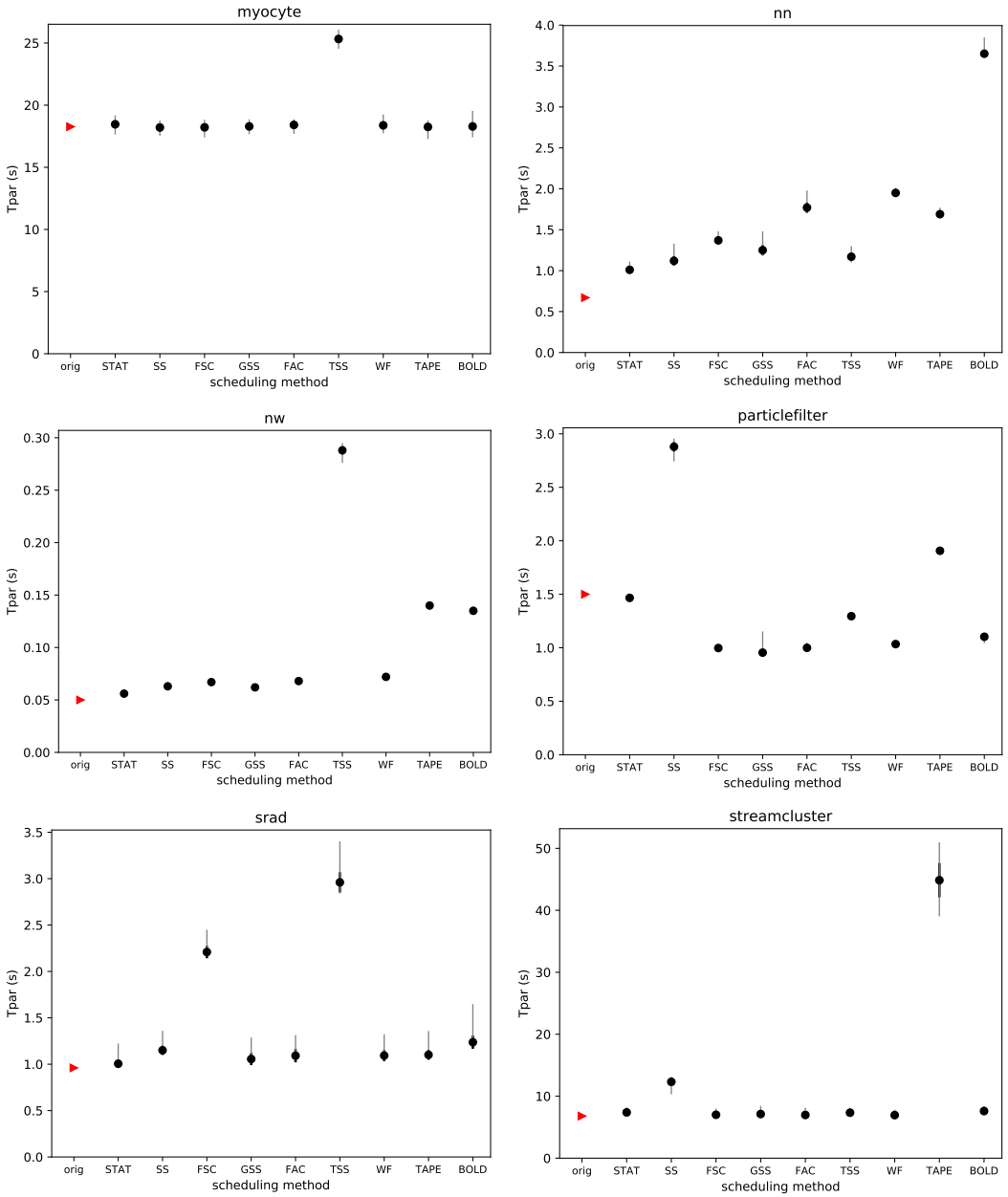

Figure 6.3: Medians, standard deviation, minimum and maximum values of the execution time of 100 runs of the Rodinia benchmarks.

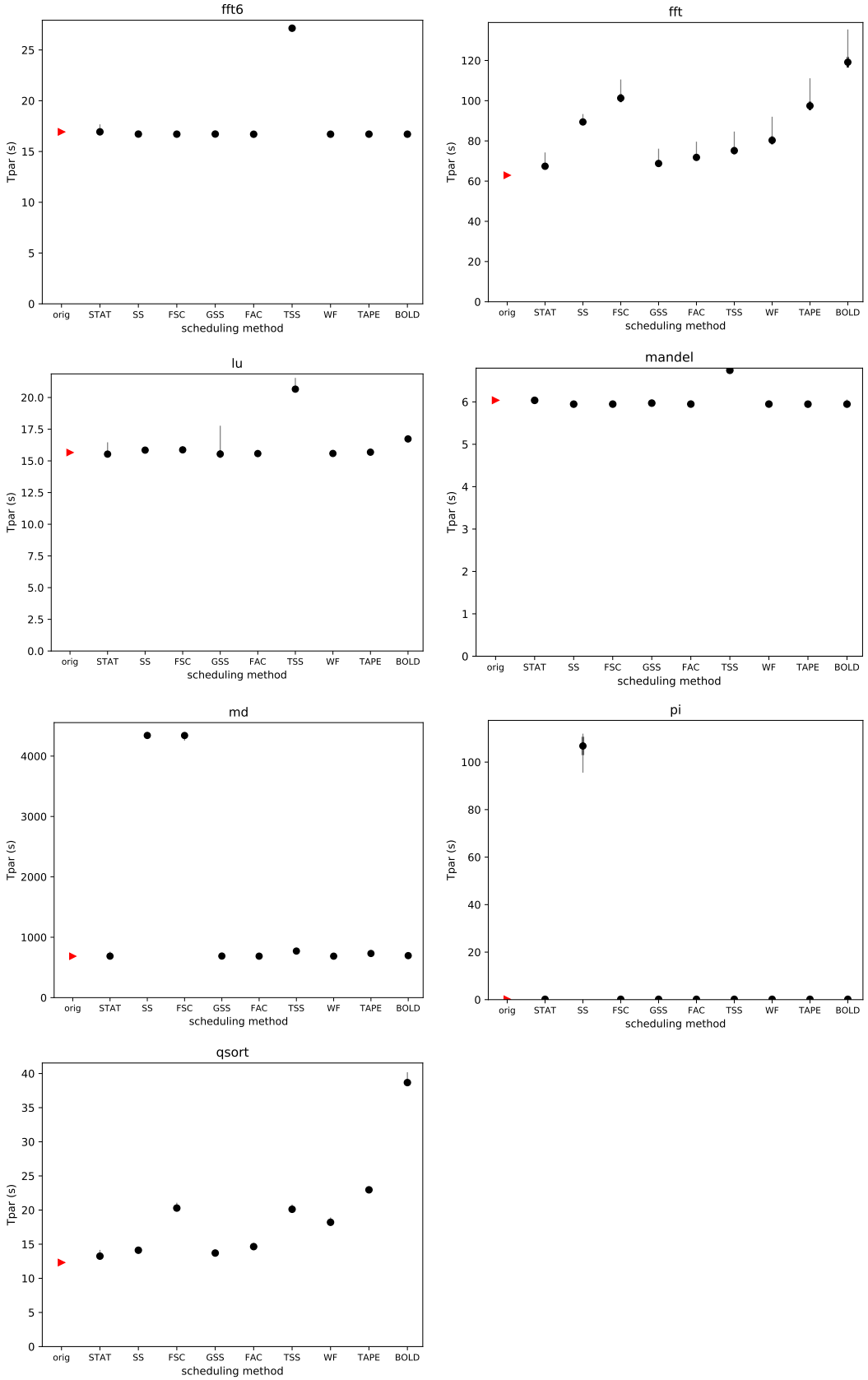

Figure 6.4: Medians, standard deviation, minimum and maximum values of the execution time of 60 runs of OMPSCR.

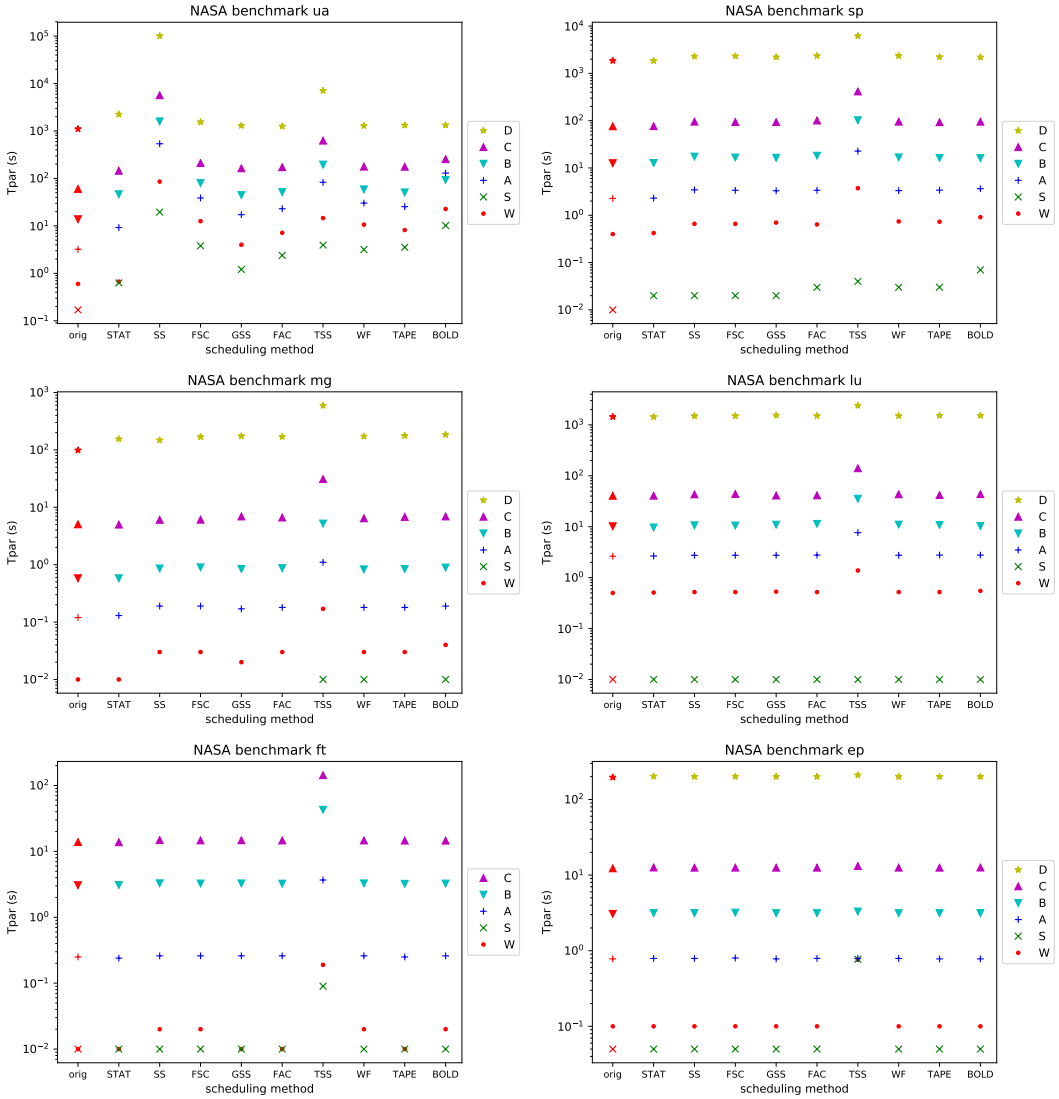

Figure 6.5: Results of the runs of one NAS benchmark with different input sizes and DLS methods.

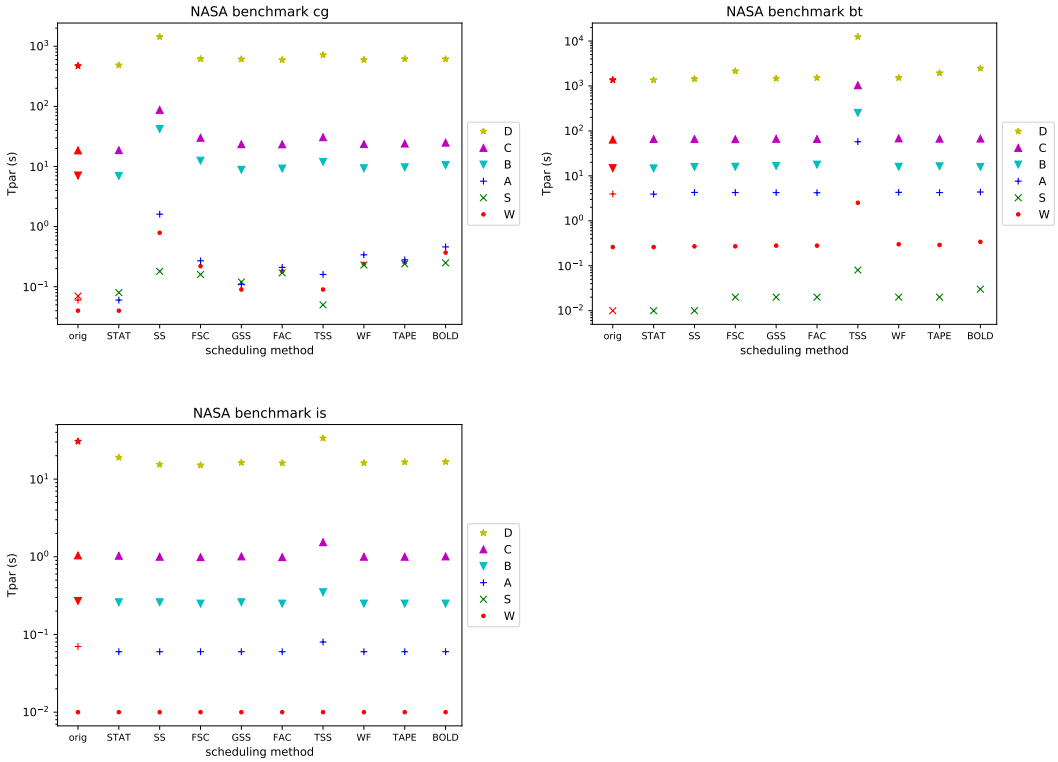

Figure 6.6: Results of the runs of one NAS benchmark with different input sizes and DLS methods.

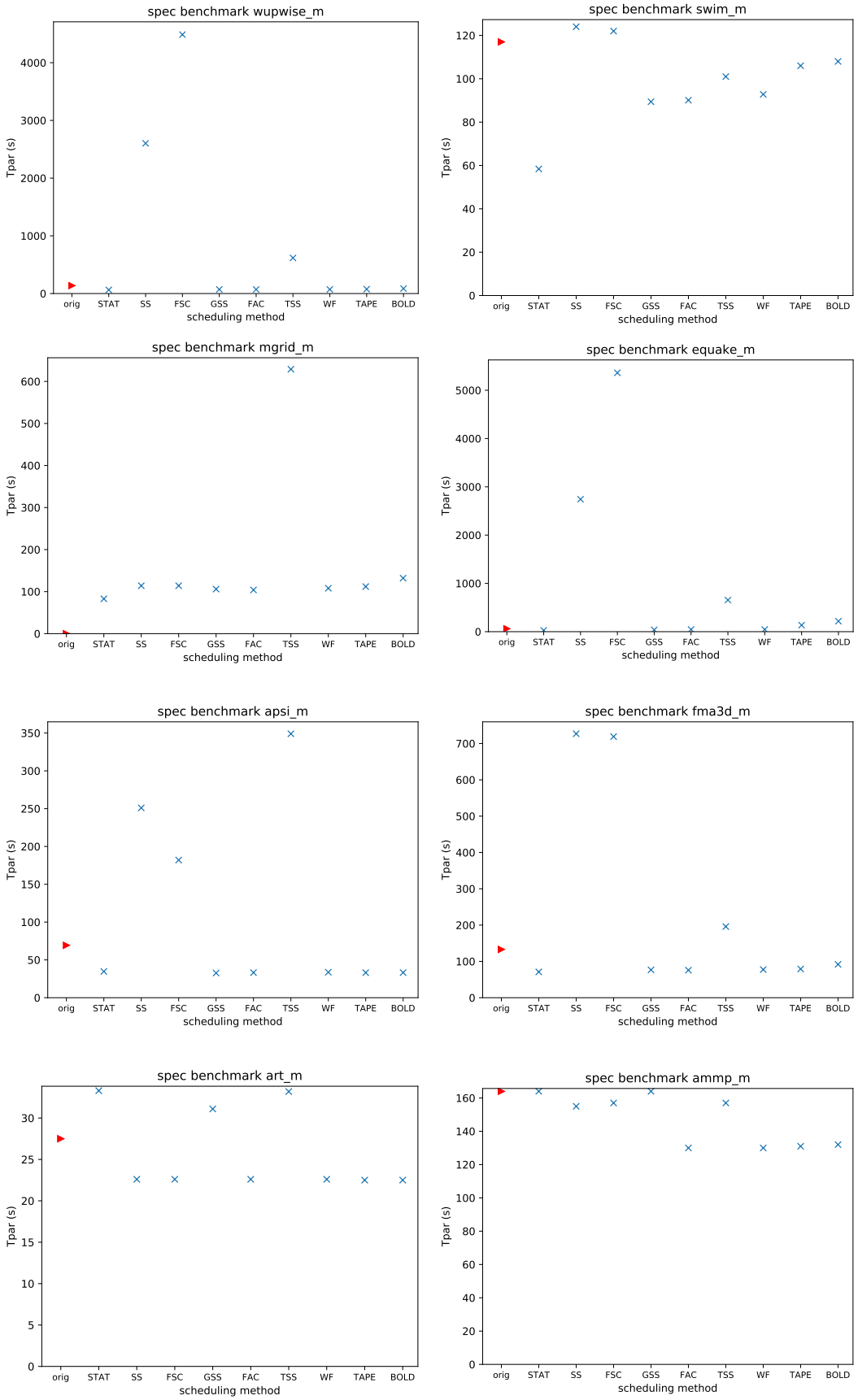

Figure 6.7: Results of the SPEC OMP benchmark runs with different input sizes and DLS methods.

7

Conclusion and Future Work

In this thesis we show that implementing new DLS methods into OpenMP is possible and feasible. These DLS methods achieved a better load balance than the static method in all experiments. Their mean and median execution times usually lie between the two extremes, static and SS, their exact position depending on the irregularity of the application, the amount of loop iterations, the single loop iteration time and the choice of input values.
When looking at the results, every one of the six implemented DLS methods managed to speed up at least one benchmark more than all the other methods. Some methods achieved a minimum execution time, others had the best mean or median execution times over all the experiments.
In the benchmarks with regular load, the factoring methods achieved similar results to the static method, since their overhead is similarly small. In benchmarks with very few iterations, the factoring methods performed worse due to their batching nature.
The most volatile methods are the ones relying on an user input. If the input is chosen well, they outperform the other methods. For less well chosen inputs, they are usually the worst performers. Especially for the BOLD and TSS methods our results confirm this. Since the input for the FSC, BOLD and TAPER methods is the same, one would expect them to yield similar results. But every one of these methods uses their input values in a different way as explained in chapter 2. Looking at each method in detail, the FSC method performed most closely to the SS method. The implementation of those two methods in libgomp is almost identically. If we set the chunk size for SS to the same value, that FSC uses, the two methods would be identical.
The TAPER method is derived from the GSS method and shows similar results. It surpasses GSS on loops with a high standard deviation of iteration execution time, but only increases the overhead time on loops with near zero standard deviation of iteration execution time. The input values need to be chosen carefully to achieve optimal performance.
The most volatile method, BOLD, is our only adaptive method. Its performance relies on not only the input of the mean and standard deviation of iteration execution time, but also on a measurement of time inside the method itself.
If each iteration takes too little time, the time measurements are not accurate enough and the method performs suboptimal. The input for the TSS method differs from the one for the

other methods. It can not be measured beforehand, but could be calculated with knowledge of the execution time of every iteration, the number of processes, the number of iterations and the underlying system status. Tzen and Mi suggest in their paper introducing TSS to set the starting value of $\frac{N}{2*P}$ and the end value of anything between the starting value and one [5]. In this thesis we generally used these values as input, but the results were not always as desired.

7.1 Future work

With this thesis we aimed to lay a foundation for future research into the use of different DLS methods. We showed that implementing those into OpenMP is possible. The results show, that the implementation is feasible. Further research could be done on the DLS methods to find an area of application for each one. Other areas would be the implementation of yet more DLS into OpenMP. For example the adaptive factoring method could be implemented by changing the GCC, not just libgomp. For it we would need to find a point inside of the GCC code, which gets executed every loop iteration to measure single loop iteration times. When this is done, the rest of the method can be implemented in libgomp. Another area of interest is the choice of DLS for a specific task. Automatically choosing the best scheduling method for the task at hand could also be an area, which future students could explore.
Some of the DLS methods considered and evaluated in this thesis did not show their full potential in the executed experiments. For example the whole point of WF is to run on machines with PUs of differing speeds. One could also vary the input values of the DLS methods to try to achieve better performance on the same experimental setup as we used. For more experiments, the papers used in chapter 2 could be consulted. It would be very interesting to see, if the libgomp implementations can be compared to the performance shown in the simulations used in these papers.

Bibliography

[1] OpenMP FAQ Page. http://www.openmp.org/about/openmp-faq/#OMPAPI. General. Accessed: 2017-10-30.

[2] Tang, P. and Yew, P.-C. Processor Self-Scheduling for Multiple-Nested Parallel Loops. In *ICPP* (1986).

[3] Kruskal, C. P. and Weiss, A. Allocating Independent Subtasks on Parallel Processors. *IEEE Transactions on Software Engineering*, SE-11(10):1001–1016 (1985).

[4] Polychronopoulos, C. D. and Kuck, D. J. Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers. *IEEE Transactions on Computers*, C-36(12):1425–1439 (1987).

[5] Tzen, T. H. and Ni, L. M. Trapezoid self-scheduling: a practical scheduling scheme for parallel compilers. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):87–98 (1993).

[6] Hummel, S. F., Schonberg, E., and Flynn, L. E. Factoring: a practical and robust method for scheduling parallel loops. In *Proc. ACM/IEEE Conf. Supercomputing (Supercomputing '91)*, pages 610–632 (1991).

[7] Hummel, S. F., Schmidt, J., Uma, R. N., and Wein, J. Load-Sharing in Heterogeneous Systems via Weighted Factoring. In *in Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 318–328 (1997).

[8] Lucco, S. A Dynamic Scheduling Technique for Irregular Parallel Programs. In *PLDI* (1992).

[9] Hagerup, T. *Allocating independent tasks to parallel processors: An experimental study*, pages 1–33. Springer Berlin Heidelberg, Berlin, Heidelberg (1996). URL http://dx.doi.org/10.1007/BFb0030094.

[10] Penna, P. H., Castro, M., Freitas, H. C., Broquedis, F., and Méhaut, J.-F. Design methodology for workload-aware loop scheduling strategies based on genetic algorithm and simulation. *Concurrency and Computation: Practice and Experience* (2016). URL https://hal.archives-ouvertes.fr/hal-01354028.

[11] Penna, P. H., Inacio, E. C., Castro, M., Plentz, P., Freitas, H. C., Broquedis, F., and Méhaut, J.-F. Assessing the Performance of the SRR Loop Scheduler with Irregular Workloads. *Procedia Computer Science*, 108(Supplement C):255 – 264 (2017).

URL http://www.sciencedirect.com/science/article/pii/S187705091730830X. International Conference on Computational Science, ICCS 2017, 12-14 June 2017, Zurich, Switzerland.

[12] Penna, P. C., Castro, M., Plentz, P. C., Freitas, H. C., Broquedis, F., and Méhaut, J.-F. BinLPT: A Novel Workload-Aware Loop Scheduler for Irregular Parallel Loops. In *WSCAD 2017 - XVIII Simpósio em Sistemas Computacionais de Alto Desempenho*, page 12. Campinas - Sao Paulo, Brazil (2017). URL https://hal.archives-ouvertes.fr/hal-01596244.

[13] Durand, M., Broquedis, F., Gautier, T., and Raffin, B. *An Efficient OpenMP Loop Scheduler for Irregular Applications on Large-Scale NUMA Machines*, pages 141–155. Springer Berlin Heidelberg, Berlin, Heidelberg (2013). URL https://doi.org/10.1007/978-3-642-40698-0_11.

[14] Che, S., Sheaffer, J. W., Boyer, M., Szafaryn, L. G., Wang, L., and Skadron, K. A Characterization of the Rodinia Benchmark Suite with Comparison to Contemporary CMP Workloads. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'10)*, IISWC '10, pages 1–11. IEEE Computer Society, Washington, DC, USA (2010). URL http://dx.doi.org/10.1109/IISWC.2010.5650274.

[15] Dorta, A. J., Rodriguez, C., Sande, F. d., and Gonzalez-Escribano, A. The OpenMP Source Code Repository. In *Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network-Based Processing*, PDP '05, pages 244–250. IEEE Computer Society, Washington, DC, USA (2005). URL https://doi.org/10.1109/EMPDP.2005.41.

[16] Bailey, D. H., Barszcz, E., Barton, J. T., Browning, D. S., Carter, R. L., Dagum, L., Fatoohi, R. A., Frederickson, P. O., Lasinski, T. A., Schreiber, R. S., Simon, H. D., Venkatakrishnan, V., and Weeratunga, S. K. The NAS Parallel Benchmarks&Mdash;Summary and Preliminary Results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, pages 158–165. ACM, New York, NY, USA (1991). URL http://doi.acm.org/10.1145/125826.125925.

[17] Feng, H., Van der Wijngaart, R., and Biswas, R. Unstructured Adaptive Meshes: Bad for Your Memory? *Appl. Numer. Math.*, 52(2-3):153–173 (2005). URL http://dx.doi.org/10.1016/j.apnum.2004.08.029.

[18] A. Frumkin, M. and Shabano, L. Arithmetic Data Cube as a Data Intensive Benchmark (2003).

[19] Spec OMP2001 FAQ page. https://www.spec.org/omp2001/docs/faq.html. Accessed: 2017-11-03.

A
Appendix

A.1 modified code of libgomp

```
case GFS_FACT:
{
if (num_threads == 0)
{
struct gomp_thread *thr = gomp_thread ();
struct gomp_team *team = thr->ts.team;
num_threads = (team != NULL) ? team->nthreads : 1;
}
ws->globalfactcounter = -1;
ws->maxworkload = 0;
ws->maxworkload = (end-start);
break;
}
case GFS_TAPE:
{
if (num_threads == 0)
{
struct gomp_thread *thr = gomp_thread ();
struct gomp_team *team = thr->ts.team;
num_threads = (team != NULL) ? team->nthreads : 1;
}
double sigma = 0;
double mue = 0;
double alpha = 1.3;
sigma = strtod(getenv("SIGMA"),NULL);
alpha = strtod(getenv("ALPHA"),NULL);
mue = strtod(getenv("MUE"),NULL);
if (mue == 0) mue = 0.0000000001;
```

```
29 ws->va = (alpha*sigma)/mue;
30 break;
31 }
32 case GFS_WFAC:
33 {
34 if (num_threads == 0)
35 {
36 struct gomp_thread *thr = gomp_thread ();
37 struct gomp_team *team = thr->ts.team;
38 num_threads = (team != NULL) ? team->nthreads : 1;
39 }
40 ws->globalWFACcounter = -1;
41 ws->WFACworkload = 0;
42 ws->WFACarray = (double*)malloc(sizeof(double) * num_threads);
43 for (int i= 0; i < num_threads; i++) ws->WFACarray[i] = 1;
44 char* env_weights = getenv("WEIGHTS");
45 for (int i = 0; i < num_threads; i++) {
46 ws->WFACarray[i] = strtod(env_weights, &env_weights);
47 }
48 ws->WFACworkload = (end-start);
49
50 break;
51 }
52 case GFS_BOLD:
53 {
54 if (num_threads == 0)
55 {
56 struct gomp_thread *thr = gomp_thread ();
57 struct gomp_team *team = thr->ts.team;
58 num_threads = (team != NULL) ? team->nthreads : 1;
59 }
60 double boldsigma = 0;
61 ws->totalspeed = 0;
62 boldsigma = strtod(getenv("SIGMA"),NULL);
63 ws->boldmue = strtod(getenv("MUE"),NULL);
64 ws->boldh = strtod(getenv("BOLDH"),NULL);
65 if(ws->boldmue == 0) ws->boldmue = 0.0000000001;
66 ws->boldtset=false;
67 ws->bolda = 2*((boldsigma/ws->boldmue)*(boldsigma/ws->boldmue));
68 ws->boldb = 8*ws->bolda*log(8*ws->bolda);
69 if (ws->boldb > 0) ws->ln_b = log(ws->boldb);
70 ws->p_inv = 1.0/num_threads;
71 ws->c1 = ws->boldh/(ws->boldmue*log(2));
```

```
72 ws->c2 = sqrt(2*M_PI)*ws->c1;
73 ws->c3 = log(ws->c2);
74 ws->boldm = end;
75 ws->boldn = end;
76 ws->boldarray = (double*)malloc(sizeof(double) * num_threads);
77 ws->speedarray = (double*)malloc(sizeof(double) * num_threads);
78 ws->timearray = (struct timespec*)malloc(sizeof(struct timespec) *
       num_threads);
79 struct timespec timerhelper;
80 clock_gettime(CLOCK_MONOTONIC, &timerhelper);
81 clock_gettime(CLOCK_MONOTONIC, &ws->boldtime);
82 for (int i = 0; i < num_threads; i++){ ws->boldarray[i] = 0;}
83 for (int i = 0; i < num_threads; i++){ ws->speedarray[i] = 0;}
84 for (int i = 0; i < num_threads; i++){ ws->timearray[i] =
      timerhelper;}
85 break;
86 }
87 case GFS_TRAP:
88 {
89 if (num_threads == 0)
90 {
91 struct gomp_thread *thr = gomp_thread ();
92 struct gomp_team *team = thr->ts.team;
93 num_threads = (team != NULL) ? team->nthreads : 1;
94 }
95 ws->trapcounter = -1;
96 ws->decr_delta = 0;
97 ws->startsize = 0;
98 start = ws->next;
99 end = ws->end;
100 chunk_size = ws->chunk_size;
101 ws->startsize = atoi(getenv("TRAPSTART"));
102 int endsize = atoi(getenv("TRAPEND"));
103 int big_n = (2*end)/(ws->startsize+endsize);
104 if (big_n != 1){ ws->decr_delta = (double) ((double) ws->startsize
       - (double) endsize)/( (double) big_n - 1.0);}
105 else ws->decr_delta = 1;
106 break;
107 }
108 case GFS_FSC:
109 {
110 if (num_threads == 0)
111 {
```

```
112 struct gomp_thread *thr = gomp_thread ();
113 struct gomp_team *team = thr->ts.team;
114 num_threads = (team != NULL) ? team->nthreads : 1;
115 }
116 double h = strtod(getenv("FSCH"),NULL);
117 double sigma = strtod(getenv("SIGMA"),NULL);
118 if (sigma == 0) sigma = 0.0000000001;
119 double test = (sqrt(2.0)*(double)end*h)/(sigma*(double)num_threads
    *sqrt(log((double)num_threads)));
120 ws->fscsize = pow(test,(2.0/3.0));
121 if (ws->fscsize > end) ws->fscsize = end;
122 if (ws->fscsize < 1) ws->fscsize = 1;
123 break;
124 }
```

Listing A.1: gomp_loop_init inside of loop.c

```
1  int globalfactcounter;
2  int maxworkload;
3  double va;
4  int globalWFACcounter;
5  int WFACworkload;
6  double *WFACarray;
7  double boldmue;
8  double boldh;
9  double bolda;
10 double boldb;
11 double ln_b;
12 double p_inv;
13 double c1;
14 double c2;
15 double c3;
16 long long boldm;
17 long long boldn;
18 long long totalspeed;
19 struct timespec boldtime;
20 double *boldarray;
21 double *speedarray;
22 struct timespec *timearray;
23 int trapcounter;
24 double decr_delta;
25 int startsize;
26 //int WFACinit;
27 int fscsize;
```

```
  int afacounter;
  double *afacarray;
  double *afacspeed;
  double *afacmu;
  double *afacsigma;
  long *afactimetaken;
  int *afacitercount;
  bool boldtset;
```

Listing A.2: gomp_work_share inside of libgomp.h

```
bool
gomp_iter_fact_next (long *pstart, long *pend)
{
struct gomp_thread *thr = gomp_thread ();
struct gomp_work_share *ws = thr->ts.work_share;
struct gomp_team *team = thr->ts.team;
unsigned long nthreads = team ? team->nthreads : 1;
long start, end, nend;
unsigned long chunk_size;
start = ws->next;
end = ws->end;
chunk_size = ws->chunk_size;
int mycounter =__sync_add_and_fetch(&ws->globalfactcounter, 1);
while (1)
{
unsigned long n, q;
long tmp;

if (start == end)
return false;
n = (end - start);
int blatest = (mycounter/nthreads)+1;
q = ceil((double) ws->maxworkload /(pow(2, blatest)* nthreads));
if (q < chunk_size)
q = chunk_size;
if (__builtin_expect (q <= n, 1))
nend = start + q;
else
nend = end;

tmp = __sync_val_compare_and_swap (&ws->next, start, nend);
if (__builtin_expect (tmp == start, 1)){
break;}
```

```
34 start = tmp;
35 }
36
37 *pstart = start;
38 *pend = nend;
39 return true;
40 }
41
42 bool
43 gomp_iter_tape_next (long *pstart, long *pend)
44 {
45 struct gomp_thread *thr = gomp_thread ();
46 struct gomp_work_share *ws = thr->ts.work_share;
47 struct gomp_team *team = thr->ts.team;
48 unsigned long nthreads = team ? team->nthreads : 1;
49 long start, end, nend;
50 unsigned long chunk_size;
51 start = ws->next;
52 end = ws->end;
53 va = (alpha*sigma)/mue;*/
54 chunk_size = ws->chunk_size;
55 while (1)
56 {
57 unsigned long n;
58 double q;
59 long tmp;
60 double Ti = (double)(end - start)/(double)nthreads;
61 if (start == end)
62 return false;
63 n = (end - start);
64 q =  (Ti + ((ws->va*ws->va)/2.0) - (ws->va*sqrt(2.0*Ti+((ws->va*ws
    ->va)/4.0))));
65 if (q<=1) q=1;
66 if (q < chunk_size)
67 q = chunk_size;
68 if (__builtin_expect (q <= n, 1))
69 nend = start + q;
70 else
71 nend = end;
72 tmp = __sync_val_compare_and_swap (&ws->next, start, nend);
73 if (__builtin_expect (tmp == start, 1)){
74 break;}
75 start = tmp;
```

```
76 }
77 *pstart = start;
78 *pend = nend;
79 return true;
80
81 }
82
83 bool
84 gomp_iter_wfac_next (long *pstart, long *pend)
85 {
86 struct gomp_thread *thr = gomp_thread ();
87 struct gomp_work_share *ws = thr->ts.work_share;
88 struct gomp_team *team = thr->ts.team;
89 unsigned long nthreads = team ? team->nthreads : 1;
90 long start, end, nend, incr;
91 unsigned long chunk_size;
92 start = ws->next;
93 end = ws->end;
94 incr = ws->incr;
95 int tempcounter =__sync_add_and_fetch(&ws->globalWFACcounter, 1);
96 chunk_size = ws->chunk_size;
97 while (1)
98 {
99
100 unsigned long n, q;
101 long tmp;
102
103 if (start == end)
104 return false;
105 n = (end - start) / incr;
106 int blatest = (tempcounter/nthreads)+1;
107 q = ceil((ws->WFACarray[omp_get_thread_num()])*((ws->WFACworkload)
      / (pow(2,blatest) * nthreads)));
108 if (q < chunk_size)
109 q = chunk_size;
110 if (__builtin_expect (q <= n, 1))
111 nend = start + q * incr;
112 else
113 nend = end;
114 tmp = __sync_val_compare_and_swap (&ws->next, start, nend);
115 if (__builtin_expect (tmp == start, 1)){
116 break;}
117 start = tmp;
```

```
118 }
119 *pstart = start;
120 *pend = nend;
121 return true;
122 }
123 double diff(struct timespec start, struct timespec end)
124 {
125 struct timespec temp;
126 if ((end.tv_nsec-start.tv_nsec)<0) {
127 temp.tv_sec = end.tv_sec-start.tv_sec-1;
128 temp.tv_nsec = 1000000000+end.tv_nsec-start.tv_nsec;
129 } else {
130 temp.tv_sec = end.tv_sec-start.tv_sec;
131 temp.tv_nsec = end.tv_nsec-start.tv_nsec;
132 }
133 double time = temp.tv_sec + (double)(temp.tv_nsec)*0.000000001;
134 return time;
135 }
136
137 pthread_mutex_t locktime;
138 bool
139 gomp_iter_bold_next (long *pstart, long *pend)
140 {
141 struct gomp_thread *thr = gomp_thread ();
142 struct gomp_work_share *ws = thr->ts.work_share;
143 struct gomp_team *team = thr->ts.team;
144 unsigned long nthreads = team ? team->nthreads : 1;
145 long start, end, nend;
146 unsigned long chunk_size;
147 start = ws->next;
148 end = ws->end;
149 struct timespec boldt = ws->boldtime;
150 pthread_mutex_lock(&locktime);
151 clock_gettime(CLOCK_MONOTONIC, &ws->boldtime);
152 pthread_mutex_unlock(&locktime);
153 struct timespec boldt2;
154 clock_gettime(CLOCK_MONOTONIC, &boldt2);
155 __sync_sub_and_fetch(&ws->boldm, (long long) ws->boldarray[
    omp_get_thread_num()]);
156 __sync_sub_and_fetch(&ws->totalspeed, (long long)ws->speedarray[
    omp_get_thread_num()]);
157 chunk_size = ws->chunk_size;
158 while (1)
```

```
159 {
160 unsigned long n;
161 double q;
162 long tmp;
163 if (start == end)
164 return false;
165 n = (end - start);
166 q = n/nthreads;
167 if(q<=1){
168 q=1;
169 }
170 else{
171 double r = 0;
172 if (ws->boldn >=0){
173 r = max(n,ws->boldn);
174 }
175 else r = n;
176 double t = ws->p_inv * r;
177 double ln_Q = log(q);
178 double v = q/(ws->boldb+q);
179 double d = n/(1.0+(1.0/ln_Q)-v);
180 if(d<=ws->c2){
181 q = t;
182 }
183 else{
184 double s = ws->bolda*(log(d)-ws->c3)*(1.0+(ws->boldm/(double)(r*
    nthreads)));
185 double w = 0;
186 if (ws->boldb > 0) w = log(v*ln_Q)+ws->ln_b;
187 else w = log(ln_Q);
188 q = min(t+max(0,ws->c1*w)+(s/2.0)-sqrt(s*(t+(s/4.0))),t);
189 }
190 }
191 if (q < 1.0)
192 q = chunk_size;
193 if (__builtin_expect (q <= n, 1))
194 nend = start + q;
195 else
196 nend = end;
197 tmp = __sync_val_compare_and_swap (&ws->next, start, nend);
198 if (__builtin_expect (tmp == start, 1)){
199 ws->boldarray[omp_get_thread_num()] = q;
200 double diff1 = diff(boldt,boldt2);
```

```
201 if (!ws->boldtset){
202 diff1 = 0;
203 ws->boldtset = true;
204 }
205 double diff2 = diff(ws->timearray[omp_get_thread_num()], boldt2);
206 __sync_sub_and_fetch(&ws->boldn, (long long) (diff1*ws->totalspeed
    +q-((diff2*q)/(q*ws->boldmue+ws->boldh))));
207 ws->totalspeed += q/((q*ws->boldmue)+ws->boldh);
208 ws->speedarray[omp_get_thread_num()] = q/((q*ws->boldmue)+ws->
    boldh);
209 ws->timearray[omp_get_thread_num()] = boldt2;
210 break;}
211 start = tmp;
212 }
213 *pstart = start;
214 *pend = nend;
215 return true;
216 }
217
218 bool
219 gomp_iter_trap_next (long *pstart, long *pend)
220 {
221 struct gomp_thread *thr = gomp_thread ();
222 struct gomp_work_share *ws = thr->ts.work_share;
223 long start, end, nend;
224 unsigned long chunk_size;
225 start = ws->next;
226 end = ws->end;
227 chunk_size = ws->chunk_size;
228 int mynumber = __sync_add_and_fetch(&ws->trapcounter, 1);
229 while (1) {
230 unsigned long n, q;
231 long tmp;
232 if (start == end) return false;
233 q = 0;
234 if(ws->startsize != 0 && ws->decr_delta !=0) {
235 q = ws->startsize - (ws->decr_delta * mynumber);
236 }
237 n = (end - start);
238 if (q < chunk_size)
239 q = chunk_size;
240 if (__builtin_expect (q <= n, 1)) nend = start + q;
241 else nend = end;
```

```
242 tmp = __sync_val_compare_and_swap (&ws->next, start, nend);
243 if (__builtin_expect (tmp == start, 1)) {
244 break;}
245 start = tmp;
246 }
247 *pstart = start;
248 *pend = nend;
249 return true;
250 }
251
252 bool
253 gomp_iter_fsc_next (long *pstart, long *pend)
254 {
255 struct gomp_thread *thr = gomp_thread ();
256 struct gomp_work_share *ws = thr->ts.work_share;
257 long start, end, nend, chunk, incr;
258 end = ws->end;
259 incr = ws->incr;
260 chunk = ws->fscsize;
261 if (__builtin_expect (ws->mode, 1))
262 {
263 long tmp = __sync_fetch_and_add (&ws->next, chunk);
264 if (incr > 0)
265 {
266 if (tmp >= end)
267 return false;
268 nend = tmp + chunk;
269 if (nend > end)
270 nend = end;
271 *pstart = tmp;
272 *pend = nend;
273 return true;
274 }
275 else
276 {
277 if (tmp <= end)
278 return false;
279 nend = tmp + chunk;
280 if (nend < end)
281 nend = end;
282 *pstart = tmp;
283 *pend = nend;
284 return true;
```

```
285 }
286 }
287
288 start = ws->next;
289 while (1)
290 {
291 long left = end - start;
292 long tmp;
293
294 if (start == end)
295 return false;
296
297 if (incr < 0)
298 {
299 if (chunk < left)
300 chunk = left;
301 }
302 else
303 {
304 if (chunk > left)
305 chunk = left;
306 }
307 nend = start + chunk;
308
309 tmp = __sync_val_compare_and_swap (&ws->next, start, nend);
310 if (__builtin_expect (tmp == start, 1)){
311 break;
312 }
313 start = tmp;
314 }
315 *pstart = start;
316 *pend = nend;
317 return true;
318 }
```

Listing A.3: iter_x_next inside of iter.c

A.2 additional table

Table A.1: Table of experiments done with respective input values for FSC, TAPER, BOLD and TSS

Benchmark Name	Total # of parallel loops	input values μ, σ, TSS_{start}-TSS_{end}
b+tree	1	$\mu = 0.000001$, $\sigma = 0.000006$, 200-10
cfd	5	$\mu_1 = 0.00000001$, $\sigma_1 = 0$, $\mu_2 = 0.00000001$, $\sigma_2 = 0.000002$, $\mu_3 = 0.00001$, $\sigma_3 = 0.00005$, $\mu_4 = 0.00001$, $\sigma_4 = 0.00004$, $\mu_5 = 0.000013$, $\sigma_5 = 0.00004$, 1000-200
hotspot	1	$\mu = 0.00011$, $\sigma = 0.005$, 200-10
hotspot3D	1	$\mu = 0.0004$, $\sigma = 0.002$, 20-2
kmeans	1	$\mu = 0.3$, $\sigma = 1.7$, 300-50
lavaMD	1	$\mu = 0.001$, $\sigma = 0.008$, 20-1
leukocyte	3	find_ellipse: $\mu = 0.0042$, $\sigma = 0.0179$, track_ellipse: $\mu = 0.15$, $\sigma = 0.66$, 20-1
lud	2	$\mu_1 = 0.000024$, $\sigma_1 = 0.000101$, $\mu_2 = 0.000002$, $\sigma_2 = 0.000015$, 20-1
myocyte	1	$\mu = 0.013$, $\sigma = 0.057$, 20-1
nn	1	$\mu = 0.000008$, $\sigma = 0.000032$, 20-1
nw	2	$\mu_1 = 0.000004$, $\sigma_1 = 0.000018$, $\mu_2 = 0.000002$, $\sigma_2 = 0.00001$, 100-5
particlefilter	10	$\mu_{1-3} = 0.0000001$, $\sigma_{1-3} = 0.000001$, $\mu_4 = 0.000001$, $\sigma_4 = 0.000004$, $\mu_{5-9} = 0.00000001$, $\sigma_{5-9} = 0.0000001$, $\mu_10 = 0.000004$, $\sigma_10 = 0.00002$, 100-5
srad	2	$\mu_1 = 0.00002$, $\sigma_1 = 0.00007$, $\mu_2 = 0.00000001$, $\sigma_2 = 0$, 50-10
streamcluster	2	$\mu_1 = 0.000001$, $\sigma_1 = 0.000004$, $\mu_2 = 0.00000001$, $\sigma_2 = 0.000001$, 200-5
fft6	4	$\mu_{1-3} = 0.0005$, $\sigma_{1-3} = 0.000005$, $\mu_4 = 6.334337$, $\sigma_4 = 26.949404$, 5-1
qsort	1	$\mu = 0.00005$, $\sigma = 0.001$, 200-10
md	2	$\mu_1 = 0.00389$, $\sigma_1 = 0.0165$, $\mu_2 = 0.00000001$, $\sigma_2 = 0.000002$, 1000-10
mandel	1	$\mu = 0.000218$, $\sigma = 0.001$, 10000-100
fft	1	$\mu = 0.00000001$, $\sigma = 0.0000001$, 262144-10
lu	1	$\mu = 0.000065$, $\sigma = 0.000276$, 200-5
pi	1	$\mu = 0.00000001$, $\sigma = 0.0000001$, 10^6-10^5
UA	77 in 11 files	$\mu = 0.0005$, $\sigma = 0.000005$, 200-10
SP	32 in 10 files	$\mu = 0.0005$, $\sigma = 0.000005$, 200-10
MG	13	$\mu = 0.0005$, $\sigma = 0.000005$, 200-10
LU	45 in 15 files	$\mu = 0.0005$, $\sigma = 0.000005$, 200-10
FT	8	$\mu = 0.0005$, $\sigma = 0.000005$, 200-10
EP	1	$\mu = 0.0005$, $\sigma = 0.000005$, 200-10
CG	18	$\mu = 0.0005$, $\sigma = 0.000005$, 200-10
BT	33 in 11 files	$\mu = 0.0005$, $\sigma = 0.000005$, 200-10
IS	9	$\mu = 0.0005$, $\sigma = 0.000005$, 200-10
wupwise	16 in 10 files	$\mu = 0.0005$, $\sigma = 0.000005$, 200-10
swim	8	$\mu = 0.0005$, $\sigma = 0.000005$, 200-10
mgrid	12	$\mu = 0.0005$, $\sigma = 0.000005$, 200-10
equake	11	$\mu = 0.0005$, $\sigma = 0.000005$, 200-10
apsi	28	$\mu = 0.0005$, $\sigma = 0.000005$, 200-10
fma3d	93	$\mu = 0.0005$, $\sigma = 0.000005$, 200-10
art	1-4 static, 5 runtime	$\mu_5 = 0.0082$, $\sigma_5 = 0.057$, 62-1
ammp	7	$\mu = 0.0005$, $\sigma = 0.000005$, 200-10

Declaration on Scientific Integrity
Erklärung zur wissenschaftlichen Redlichkeit

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

Author — Autor
Patrick Buder

Matriculation number — Matrikelnummer
09-062-480

Title of work — Titel der Arbeit
Evaluation and Analysis of
Dynamic Loop Scheduling in OpenMP

Type of work — Typ der Arbeit
Master Thesis

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, 30.10.2017

Signature — Unterschrift