

Tagging for Flexible Loop Scheduling in OpenMP

Spyros Mantelos
Dept. of Computer Science and Engineering
University of Ioannina
Ioannina, Greece
s.mantelos@uoi.gr

Vassilios V. Dimakopoulos
Dept. of Computer Science and Engineering
University of Ioannina
Ioannina, Greece
dimako@cse.uoi.gr

Abstract

The OpenMP programming model provides a number of loop scheduling policies which the application developer can utilize to split loop iterations among the threads executing a parallel region. Because determining the optimal schedule policy and its parameters may be a tedious task, OpenMP offers a special schedule, called *runtime*; this allows the application developer to experiment with different scheduling policies at execution time, using a special environment variable, without the need to re-compile their program. However, there is a fundamental limitation to this facility: all application loops with the runtime schedule must follow the exact same scheduling policy. We propose a simple but effective extension that alleviates this limitation by allowing for different, per-loop runtime schedules. In addition, we use this mechanism to introduce extra loop scheduling policies which are not officially supported by OpenMP but may however prove effective in certain applications.

1 Introduction

OpenMP [1] is a popular application programming interface which keeps evolving. Based on compiler directives, it offers a layer of abstraction for creating parallel applications with many low level operations like thread creation handled automatically. Since its inception, one of the core functionalities of OpenMP is loop parallelization; special directives specify that the iterations of a given `for` loop be divided between all available threads. In addition, it provides three different, parametrizeable scheduling methods for controlling how the iterations will be distributed. However, finding the optimal way to divide the iterations is not a trivial task; among a variety of factors, the optimal scheduling strategy may depend on the code a loop contains and the speed or load of the cores executing its iterations.

To retain flexibility, the OpenMP API offers runtime functions for setting values that affect various aspects of the execution of an application. Additionally, some of those values can be initialized through specific environment variables. This way, there is no need to keep modifying the application code if one only wishes to experiment with different values for some specific parameter. The loop scheduling method is one of the execution parameters a user can modify at runtime. However, because a single environment variable initializes the schedule values, a single choice must be made for all the runtime-controlled loops of the program.

The three available scheduling methods is a further limitation of OpenMP. While they can be parametrized to cover a wide variety of loop-based applications, other methods have been proposed and proven to be more effective in specific cases (e.g. [2, 3]). This motivated Müller Korndörfer et al. [4] to implement additional scheduling strategies for the Clang/LLVM compiler. Their implementation utilizes the *runtime* scheduling option of OpenMP plus additional environment variables; as such their scheme still has the limitation that all runtime-controlled loops of the program use the value of the same environment variable.

In this work we first propose a novel extension for OpenMP that allows per-loop runtime selection and parametrization of the applied scheduling strategy. We achieve this by introducing a tagging mechanism which allows labeling the OpenMP constructs of interest and then controlling them through matching environment variables. We then present an infrastructure which allows the addition and utilization of new scheduling strategies, in a portable way. The new strategies may be utilized either through the OpenMP *auto* schedule and/or the aforementioned tagging mechanism. The major contributions presented in this paper are the following:

- A language-level extension that allows assigning a tag to any OpenMP loop construct either through a new clause or through a new directive.
- A runtime mechanism that provides the means to control the scheduling of loops with a given tag.
- An infrastructure for implementing and utilizing additional loop scheduling strategies for OpenMP loops in a portable way.
- A full implementation of all the above in an open-source OpenMP compiler [5].

The rest of the paper is organized as follows: After discussing related work in Section 1.1, we give an overview

of what OpenMP offers for loop scheduling, in Section 2. Section 3 presents our tagging proposal while Section 4 presents the mechanisms used to implement and deploy new scheduling policies. We then present experimental results to demonstrate and evaluate our proposal in Section 5. Finally, Section 6 concludes the paper.

1.1 Related Work

Loop scheduling algorithms has been a topic of quite extensive research and a comprehensive survey is beyond the scope of this work. We only present a small subset of well known methods here. Work is usually given out to threads in chunks of consecutive iterations. Many scheduling strategies, including the ones offered by OpenMP, focus on calculating an efficient chunk size based on a variety of criteria, while trying to keep synchronization overheads low; fixed size chunking [6], taper [7], factoring [8], and trapezoid self-scheduling [2] are characteristic examples. Weighted Factoring [9] requires additional knowledge about the relative processing unit speeds. Fractiling [10] exploits the self-similarity properties of fractals, attempting to maximize data locality.

A different approach is followed by the so-called *adaptive* scheduling methods. These policies change their scheduling decision according to data they collect during runtime; [4] is a comprehensive survey for such methods, as well as the methods mentioned above.

Another idea is improving load balance by work stealing. The affinity scheduling strategy [3] employs this technique while also accounting for processor affinity; iCH [11] is another recent scheduling proposal which uses the same technique. Finally there have been proposals of automatic scheduling method selection [12–14].

There have been some works aiming to enhance the scheduling capabilities of OpenMP, focusing mainly on the addition of extra scheduling strategies. OpenMP supports three scheduling schemes, *static*, *dynamic* and *guided* [1]. A methodology for implementing new scheduling techniques is given in [15] for the LLVM compiler infrastructure, where the authors implemented an additional scheduling scheme, while [4, 16] argue that more scheduling techniques should be included. The authors in [17] make a proposal for allowing arbitrary user-defined scheduling methods to be supported in OpenMP. Our tagging proposal in this work is orthogonal to all these works since it is schedule-agnostic and offers a way to employ different policies on different runtime-scheduled loops.

2 OpenMP Loop Scheduling

OpenMP offers three loop scheduling methods, namely *static*, *dynamic* and *guided*. In all these methods, work is assigned to threads in chunks of consecutive iterations; a *chunksize* parameter may be specified to control the size of iteration chunks. In *static* scheduling, the iteration distribution is predetermined and fixed, thus keeping the scheduling overheads small. Chunks are assigned in a round-robin fashion to the available threads. Without a *chunksize*, the iterations are divided to chunks of approximately equal size and each thread gets at most one chunk. The *static* schedule is a good choice for loops whose iterations have approximately the same amount of work to do, and the available processing units are evenly loaded. Otherwise, this scheduling strategy will lead to suboptimal performance.

The *dynamic* and *guided* methods are self-scheduling policies and assign chunks dynamically during the program execution. While *dynamic* distributes chunks of equal size, *guided* initially assigns larger chunks, in order to keep the scheduling overheads low; then the chunk sizes are gradually reduced so as to achieve better load balance. For the *guided* policy, if the user provides a *chunksize*, it will act as a lower bound on the size of chunks given away (except possibly for the last chunk which may be smaller).

OpenMP also supports two more schedule "types", *auto* and *runtime*. The former is an unspecified, implementation-defined schedule; the compiler and/or the runtime system is free to map iterations to threads in any possible way. The *runtime* schedule is not actually a different scheduling policy; it just allows picking the exact scheduling method during the program execution (instead of hard-coding it in the program source), by selecting among the four methods available (*static*, *dynamic*, *guided* and *auto*). This allows experimenting with different scheduling methods during runtime execution, without recompiling the application.

To understand the mechanism, one needs to know that an OpenMP implementation must maintain a number of internal control variables (ICVs) which control the behavior of an OpenMP program. One of the ICVs is *run-sched-var* which specifies the actually employed method when a *runtime* schedule is in effect. The initial value of *run-sched-var* is implementation-specific while its value may change at any time during a program's execution. When an OpenMP application starts its execution, a set of environment variables are checked, and if the user has assigned a value to any of them, the corresponding ICV is updated accordingly. After that, OpenMP API routines can be called to retrieve or modify the value of the ICV.

For worksharing loops parallelized by the OpenMP `for` directive, the `schedule` clause may be used to force a particular scheduling policy for that loop. This clause accepts the name of the scheduling method and, optionally, an integer that represents the desired *chunksize*. Determining the actual scheduling policy goes through the following steps:

1. If there is no `schedule` clause, the scheduling policy of the *def-sched-var* ICV is employed; this is implementation-defined and there is no way for a programmer to determine or change its value.

2. If a `schedule` clause is present and it specifies any

of the scheduling policies except *runtime*, the specified policy is used.

3. The presence of `schedule(runtime)` clause forces the scheduling policy of the *run-sched-var* ICV.

In the last case, the `OMP_SCHEDULE` environment variable can be used to give the desired initial value to the *run-sched-var* ICV. The value may also be changed programmatically through the `omp_set_schedule()` runtime API call.

There are two things to notice here:

- First, because a single environment variable controls the value of *run-sched-var*, *all* loops parallelized with the `schedule(runtime)` will be forced to use the same scheduling policy.
- Second, although the `omp_set_schedule()` can change the value of *run-sched-var* dynamically, it must specify a concrete schedule type which has to be hard-coded in the program source code; changing the schedule type thus requires re-compilation of the user program after all.

As a result, for applications that contain more than one parallelized `for` loops, OpenMP offers no easy way for exploring the scheduling policy space without repetitive source code-level changes and re-compilations. Our proposal tries to remedy this limitation in an easy and portable way.

3 Proposed Extension: Tags

In what follows, when we talk about loops, we imply loops parallelized with the OpenMP `for` directive, utilizing the `schedule(runtime)` clause. In order to overcome the limitations of the *runtime* OpenMP schedule when multiple such parallel loops are present, the user must be able to specify different scheduling policies for different loops. The single `OMP_SCHEDULE` environment variable can not provide the means to do that; the natural solution is to use a *separate* environment variable for each loop.

In order to be able to identify a loop so as to provide a specific environment variable for it, we propose a labeling scheme for loops. In particular, we propose the addition of a new `tag` clause which can be used to label a `for` or a combined `parallel for` construct, with the following syntax:

`tag(` *label* `)`
`tag(` *label*, *n* `)`

The clause accepts a legal label name to be used as the tag of the construct. Optionally, it accepts a second argument which must be an integer expression; its value will be concatenated with the first argument, forming the final label name. The second form is useful in the case the OpenMP loop of interest is enclosed within another loop; using the index of the outer loop as the second argument of the `tag` clause produces a unique label for each of the instances of the inner loop.

While the `tag` clause is a very easy way to label a loop construct, its presence results in a non-compliant OpenMP program as no compiler will accept it (save OMPi, which we have modified to accept and act on it). For this reason, we also propose a compliant and portable way of tagging *any* OpenMP construct, with the following general form:

`#pragma ompext tag(` *label* [*, n*] `)`
openmp-construct

The OpenMP construct we need to label is preceded by a non-OpenMP `tag` directive which provides the label. This pragma should be ignored by standard compilers, causing no compatibility problems.

Labeling arbitrary OpenMP constructs with the `tag` directive is quite general and, in addition, it allows considerable flexibility. For example, if a parallel region contains multiple `for` loops and we want to set the same tag for all of them, instead of using a `tag` clause on every single loop, we can set a tag for the whole parallel region. The tags are passed down to any descendant threads, so any code inside the parallel region can use the information we set for the specified tag.

A tagging directive may be nested inside an OpenMP construct that has already been labeled by another tagging directive. In the above example, some of the loops inside the tagged parallel region could have their own tags; if a loop construct has not been individually tagged, it inherits the tag assigned to the region it appears in.

3.1 Per-Tag Environment Variables

Given that loops can be identified with tags, we can specify their runtime schedule using different environment variables. The environment variable that corresponds to a loop tagged with *label* is named `OMPI_TAG_SCHED_`*label*. For example the environment variable for the tag `loop1` would be `OMPI_TAG_SCHED_loop1`. For compatibility, the value given to such an environment variable follows the standard OpenMP syntax. For example, setting `OMPI_TAG_SCHED_loop1=static,10` would make the loop tagged as `loop1` to utilize the *static* scheduling policy with a *chunksize* of 10 iterations.

An immediate observation is that since these environment variables are not bound to OpenMP rules, they could utilize an alternative syntax for their values. Taking this one step further, such variables could be used to select *additional scheduling policies*, which the OpenMP standard has not yet endorsed. This is exactly what we present in Section 4; we have implemented a number of new scheduling policies and we exploit the tagging mechanism to apply them to specific loops. Consequently, except for the standard OpenMP syntax, we allow an additional syntax for giving values to the per-tag environment variables:

OMPI_TAG_SCHED_*label* = *policy* (*arg* [, *arg*] ...)

where *arg* is a parameter-value pair for the given scheduling *policy*. We call this a *parameter-based* syntax. For example, `c` represents the *chunksize* parameter. In the above example, the *static* policy with a *chunksize* of 10 iterations could be equivallently specified in parameter-based syntax as:

```
OMPI_TAG_SCHED_loop1=static(c=10)
```

More details will be presented in Section 4.

3.2 Scope and Runtime Handling of Tags

In order to support nesting of tags, a tag stack is maintained at runtime for each implicit or explicit task. Descendant tasks inherit the ancestor's stack and may top it with new tags. For example, when a parallel region is encountered, the created threads (i.e. implicit tasks) start with the parent thread tag stack. Whenever a new tag is encountered for an OpenMP construct C, the label is pushed onto the task's tag stack. The tag's *scope* is the region of construct C; any nested constructs within C also have access to the given tag. When the execution of C is completed, the tag is popped off the stack.

When encountering a loop construct with a *runtime* schedule, the following procedure is effected[1]:

- If the loop construct is not tagged, the topmost tag in the stack whose matching environment variable provides scheduling information is utilized.
- If the loop construct is tagged, the top element of the tag stack is compared with the given label.
- If the two labels concur, the value of the matching environment variable is utilized to determine the desired schedule.
- If the two labels do not concur, or the matching environment variable has not been set by the user, a default value is used, as determined from the global ICVs.

Consider Listing 1 as an example. Let us assume that the environment variables `OMPI_TAG_SCHED_outer` and `OMPI_TAG_SCHED_nested` have been set with the values `guided` and `dynamic`, correspondingly. Finally, there is no `OMPI_TAG_SCHED_dummy` variable set. When the application executes, a tag will be created by the initial thread before reaching the parallel region (line 1). Since its corresponding environment variable has been set, its value is read and stored. After the thread team is created, all child threads contain the "outer" label on their tag stacks. When reaching the first OpenMP loop (line 4), the "outer" label will be utilized since the loop has no tag and "outer" is the topmost label on the stack; consequently the loop will use a *guided* schedule. The second loop (line 6) has a tag, and the matching environment variable has been set; it will use a *dynamic* schedule and when done, the "nested" label will be popped off the stack. The loop in line 8 has a tag whose matching environment variable has not been set; the default scheduling method of the implementation will thus be used and the "dummy" tag will be popped off the stack. The fourth loop (line 10) will use a *guided* schedule since the "outer" label is still in effect; it leaves the tag stack at the end of the parallel region.

[1] Unless the standard `OMP_SCHEDULE` environment variable is set.

Listing 1: Tag example

```
1  #pragma ompext tag("outer")
2  #pragma omp parallel
3  {
4      #pragma omp for schedule(runtime)
5      for-loop
6      #pragma omp for schedule(runtime) tag("
           nested")
7      for-loop
8      #pragma omp for schedule(runtime) tag("
           dummy")
9      for-loop
10     #pragma omp for schedule(runtime)
11     for-loop
12 }
```

3.3 Implementation in the OMPi compiler

The implementation of the tagging extension in OMPi was relatively straightforward, requiring additions to the parser grammar and the code generator. The transformation phase replaces the `tag` directive by two runtime calls that push the label name onto the tag stack and then pop it. The generated function calls are placed right before and right after the tagged OpenMP construct, correspondingly. The runtime library implements the tag stack and updates it for each task as described in 3.2.

When a label is pushed on the stack, the corresponding environmental variable is retrieved and parsed to identify the required policy and its parameters; the parameters are then stored and utilized in every loop that is tagged by that label.

4 New Scheduling Policies

It is well known that for particular classes of applications certain scheduling methods extract better performance than what the standard OpenMP schedules offer [2, 3]. Having access to such methods is a very useful tool when striving for optimal scheduling times. A setback would be the possible need to modify the compiler in order to accommodate the new schedules and their parameters at the language level. Müller Korndörfer, et al. [4] present techniques for implementing extra scheduling policies within the LLVM framework. They utilize the existing runtime interface so as not to fiddle with the transformation and code

generation stages of the compiler. Their scheme is implemented entirely within the OpenMP runtime library and the new loop schedules become available through the *runtime* schedule option of OpenMP. For selecting the new methods, the `OMP_SCHEDULE` environment variable is used, whose syntax was modified to allow additional values. For providing parameters to the new policies, new environment variables were utilized.

We take a similar approach to introducing new scheduling policies in the OMPi compiler, in that their implementation is contained entirely within the OpenMP runtime library and requires no compiler modifications. In contrast to [4] we do not exploit the *runtime* schedule mechanics; we use two different techniques, both of which allow a user to employ the new scheduling policies without breaking compatibility with the OpenMP specifications.

The first technique utilizes the *auto* schedule type which is implementation-defined anyway. We created a new environment variable to be used when *auto* is selected. This variable is called `OMPI_SCHED_AUTO` and follows the same syntax as the variables described in Section 3.1, used by the tagging mechanism.

The second way for the user to employ new scheduling policies is through our tagging mechanism. In particular, as mentioned in Section 3.1, the parameter-based syntax of the environment variables that match the labels on tagged constructs allows for richer descriptions. Each scheduling policy has a number of defining parameters; all the user has to do is state the policy name along with values for the required parameters.

Implementation-wise, the introduction of new scheduling policies involved modifications to the runtime infrastructure but did not require any changes to the compiler. During code transformations OMPi normalizes all loops and the generated code for *runtime*-scheduled loops embeds library calls for a thread to get the initial iteration chunk along with a pointer to a chunk-distributing function, called `get_next_chunk`. It then repeatedly calls `get_next_chunk`, recovers the loop indices from the normalized ones and executes the loop body until `get_next_chunk` returns no further iterations. In the runtime library, we implemented new chunk distributing functions, one per policy. Minor modifications were also needed to the scheduling method selection logic to incorporate the new functions and associate them with the tag mechanism. If the loop is tagged, the top of the tag stack is first checked for the loop's label. If the corresponding environment variable has has been defined, the stored policy parameters are retrieved and `get_next_chunk` is made to point to the matching function.

In what follows, we give a brief presentation of all the scheduling policies we have made available. Their parameters are summarized in Table 1; some of them are optional and they are marked with a "no" in the *Required* column. Certain policies require the mean and standard deviation of the iterations execution time. To help with obtaining these values, we have an additional pseudo-policy named *profiling*; the user executes the program once to get the required profiling information and then uses it to set the scheduling method's parameters accordingly, for the final run. To find the scheduling overhead, we used a micro benchmark presented in [18].

Table 1: Available scheduling policies and their parameters

Policy	Parameter	Symbol	Required
static	chunksize	c	no
dynamic	chunksize	c	no
guided	chunksize	c	no
Trapezoid	first	f	no
	last	l	no
FSC	standard deviation (σ)	s	yes
	overhead	h	yes
Taper	mean (μ)	m	yes
	standard deviation (σ)	s	yes
	c.o.v. factor	a	no
	min. chunksize	c	no
Factoring	mean (μ)	m	yes
	standard deviation (σ)	s	yes

4.1 Trapezoid Self-Scheduling

Trapezoid Self Scheduling [2] works like *guided* but uses a linear function for decreasing chunk sizes. It takes two parameters, the sizes of the first (f) and the last (l) chunks. The authors claim that the linearity leads to reduced scheduling overheads; additionally they suggest $\frac{N}{2P}$ as a good size for the first chunk, where N is the total number of iterations and P the available processing units. We used this as a default value in case the user does not specify one. The method uses the first and last chunk sizes to calculate the total number of chunks and the amount (δ) by which chunk sizes get decreased; the size of the first chunk is given by $c_1 = f$ and the size of the ith chunk is given by:

$$c_i = c_{i-1} - \delta$$

4.2 Taper

Taper [7] attempts to achieve optimal load balance and reduce scheduling overheads by calculating the largest possible chunk and decreasing the total number of chunks. Like *guided*, it distributes chunks in decreasing sizes, but takes into account the mean and standard deviation of the iteration execution times. In the following equation, u_a is the coefficient of variation ($\frac{\sigma}{\mu}$) multiplied by a scaling factor a which is found empirically, T_i is the number of remaining iterations divided by the number of processing units, and c_{min} denotes an optional lower limit for the chunk size, c_i:

$$c_i = max \left\{ c_{min}, \left\lceil T_i + \frac{u_a^2}{2} - u_a \sqrt{2T_i + \frac{u_a^2}{4}} \right\rceil \right\}$$

4.3 Fixed Size Chunking

Fixed Size Chunking [6] aims to reduce scheduling overheads by scheduling multiple iterations at a time. It is similar to *dynamic* scheduling but alleviates the user of having to find the optimal chunk size by automatically calculating it using the following formula:

$$c = \left(\frac{\sqrt{2}Nh}{\sigma P \sqrt{logP}} \right)^{2/3}$$

The required inputs are the standard deviation of iteration execution times (σ) and the method's scheduling overhead h, which according to [6] is independent of the chunk size; P is the number of available processing units and N is the total number of iterations.

4.4 Factoring

Factoring [8] is also similar to *guided* scheduling. The idea is to take into account the execution time variance to achieve an even better load balance. The difference from other methods is that it schedules iterations in batches of P equally-sized chunks (P being the number of processing units). The chunk size c_i for the ith batch is calculated by the following equation, where R_i is the number of remaining iterations and x_i is a value that depends on the coefficient of variation ($\frac{\sigma}{\mu}$):

$$c_i = \left\lceil \frac{R_i}{x_i P} \right\rceil$$

5 Evaluation

To evaluate our mechanism we use two applications from the OpenMP Source Code Repository [19] and one from the CORAL Benchmarks [20], namely *molecular dynamics*, *Bailey's "6-step" Fast Fourier Transformation* and the *XS benchmark*. The applications contain multiple parallel loops, all of which were tagged appropriately and modified so as to use the *runtime* OpenMP scheduling option. We then run each application using combinations of schedule types through our environment variables for the tagged loops and measure execution times. For each combination, we record timing results for 10 runs in total.

We performed our experiments in two systems with very different capabilities. The smaller one is based on a single 8-core Intel Xeon E5-2620v4 CPU with hyperthreading disabled, has 8 GiB of memory and runs Ububtu Linxu 18.04.5 OS. The larger one is equipped with four 16-core/32-threads Intel Xeon Gold 6130 CPUs, for a total of 64 physical cores and 128 hardware threads. This system runs on CentOS 8 and has a total amount of 256 GiB of RAM.

Each application was executed exhaustively with all schedule combinations for the loops involved, and for a variety of parameter values. Here we present only the most representative scheduling combinations, with parameters which offered the best performance for each combination. Unless otherwise specified, the default *chunksize* values are implied for the OpenMP schedules.

To avoid thread placement randomness and/or irregularities by the OS, the molecular dynamics and XS benchmarks were run by pinning threads to cores; this was achieved by setting the environment variables `OMP_PLACES=cores` and `OMP_PROC_BIND=true`. Bailey's FFT contains nested parallel regions, so in order to distribute all threads across the available cores in the 64-core system, we used `OMP_PROC_BIND=spread,close`. This leads to a sparse distribution of threads to sockets for the first level of parallelism while keeping the second level close to the primary thread.

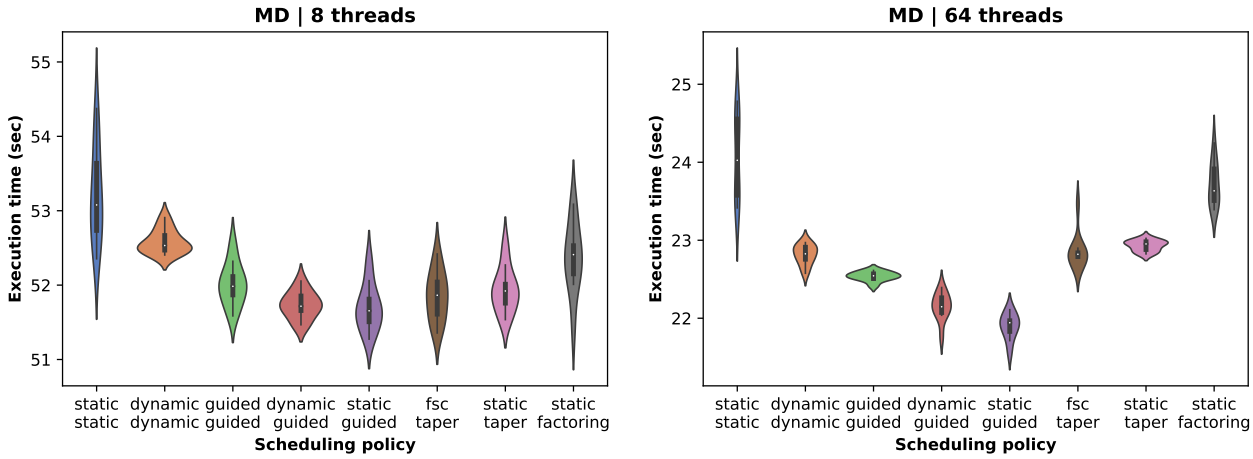
5.1 Molecular dynamics

This application is an implementation of a molecular dynamics simulation for the velocity Verlet algorithm [21]. The program requires two inputs, the number of particles and the simulation steps. For both systems we selected 20 simulation steps. For the larger system we used 65536 particles and for the smaller one 32768. It contains two parallel `for` loops which we used for our experiments.

The results are presented in Fig. 1. For each system we present a selection of policy combinations which better convey what can be expected from the attainable performance. In the plots execution times are measured in seconds while the x-axis contains the scheduling policies used for the two loops. The violin plots depict all 10 runs for each scheduling pair; the shorter the area, the less the variation in execution times. In Fig. 1(b) we can see that by using the tag directive with a combination of two policies, we achieve better times than using a single OpenMP policy. It is interesting that by combining *dynamic* or *static* with *guided* using the tagging mechanism, we get the best performance; this shows that even with the build-in scheduling policies of OpenMP, performance can be improved substantially by differently scheduled *runtime* loops. On the other hand, on the system with 8 cores, we observe no significant differences on the execution times by using the new policies, as seen on Fig. 1(a). Still, the two combinations of the basic scheduling policies, are again better than simply using one of them for both loops. In both systems, it is clear that the *static* policy is the least stable one, resulting in large variations in execution times.

5.2 XS Benchmark

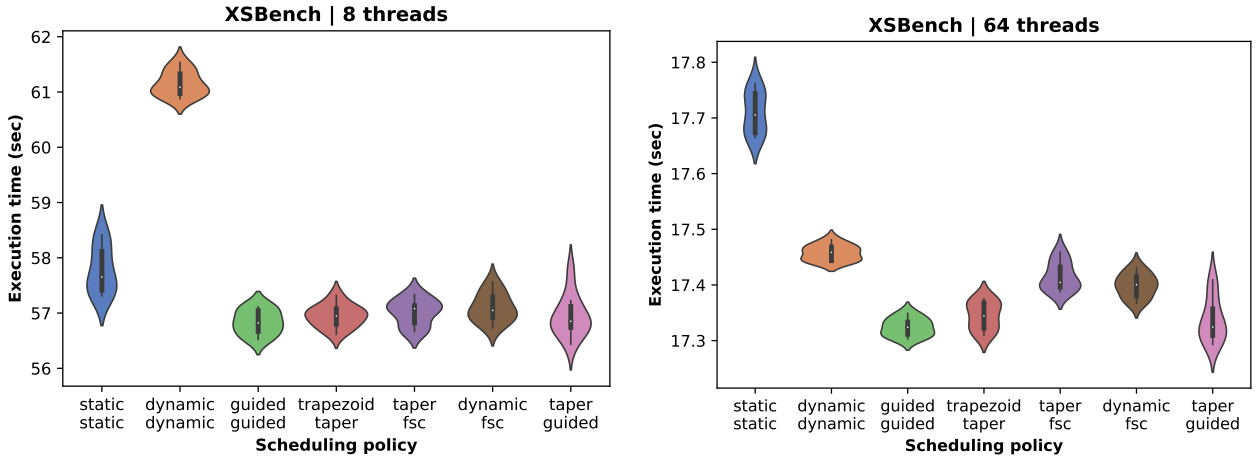
The XS benchmark models the most computationally intensive part of a typical Monte Carlo transport algorithm, and accounts for around 85% of the total execution time of OpenMC [22]. The application contains two parallel `for` loops, one in the initialization phase and one in the actual calculation. On the system with the 64 processing cores, we



(a) Using 8 threads on the smaller system

(b) Using 64 threads on the larger system

Figure 1: Molecular dynamics results



(a) Using 8 threads on the smaller system

(b) Using 128 threads on the larger system

Figure 2: XS benchmark results

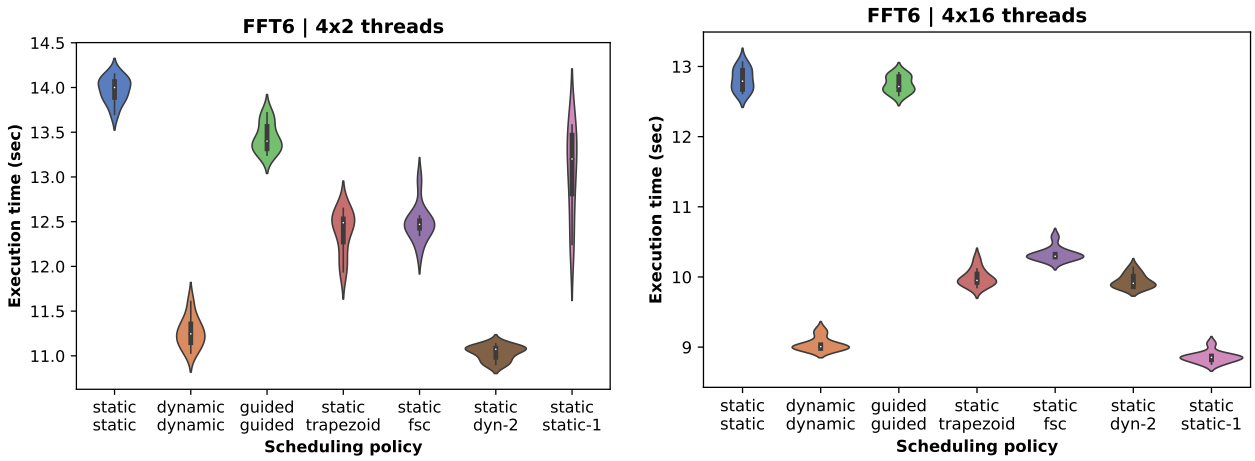
set the cross section lookups to 30 million which is double the default value.

The results are shown in Fig. 2. In both systems the best execution times were again achieved by using a combination of scheduling methods (*taper* and *guided* in particular). For *taper* we used $a = 1.3$, and $m = 6, s = 9.949$ ($m = 9, s = 35.902$) on the smaller (larger) system as obtained after a profiling session. However, by using *guided*, the execution times have a smaller variation, while achieving the same mean.

5.3 Bailey's "6-step" FFT

This application is an implementation of an FFT algorithm that consists of six steps and is presented in [23]. The program contains four `parallel for` loops, the three of which are nested inside the first one. Because the three nested loops have similar properties, we used the same tag for all of them. We thus tried combinations of two scheduling policies, the first one corresponding to the outer loop and the second one corresponding to all three inner loops. The application requires two inputs, the signal strength and the number of outer loop iterations. We selected 20 iterations for both systems. For the larger system the signal strength for our experiments was 8192 and for the smaller one 4096. Because of the nested parallel regions, care was taken so as to avoid oversubscription which was found to impact performance. On the smaller system we used a configuration of 4×2 threads (outer $\times$ inner level) while on the larger system we report results for 4×16 threads which produced the best performance.

The results are displayed in Fig. 3; in the scheduling policy labels, the first (top) name refers to the outer loop. Because of the few iterations, *guided* leads to imbalances, so



(a) Using 8 threads on the smaller system

(b) Using 64 threads on the larger system

Figure 3: Bailey's "6-step" FFT results

static or *dynamic* are proven to be the best choices for the outer loop. The iterations of the inner loops contain uneven amounts of work, so a *dynamic* (8-core system) or *static* (64-core system) schedule with a small *chunksize* is more effective; *static* with a large *chunksize* or no *chunksize* at all produces large imbalances leading to the worst possible performance. Once again, using a single scheduling policy for all loops, does not yield the best results. We can also observe that the other two basic policies, unperformed as compared to the best achievable times, showing the significance of experimenting and selecting the most appropriate scheduling technique.

6 Conclusions and Future Work

In this work we propose a tagging mechanism for OpenMP constructs which offers considerable flexibility with respect to loop scheduling. In particular, it overcomes the OpenMP limitation of a single scheduling policy for all application loops with the `schedule(runtime)` clause. The mechanism allows per-loop runtime scheduling decisions through separate environment variables. The same mechanism can also be used as a means of providing additional scheduling strategies beyond the ones offered by OpenMP, extending the arsenal of scheduling options available to programmers. Our tagging proposal was implemented in the OMPi compiler along with four new scheduling methods. The implementation was evaluated using different applications where it was found that the basic OpenMP scheduling techniques do not always distribute iterations in an optimal manner.

As part of our future plans, we will be implementing additional loop scheduling techniques. The syntax of the tag-related environment variables is quite general and should support arbitrary schedules. In another direction, we work on exploiting the tagging mechanism to provide runtime flexibility for other OpenMP constructs, as well. Our `tag` directive can be applied to any OpenMP construct, so conceivably we could offer runtime parametrization of any directive through corresponding environment variables, similar to what we did for worksharing loops. In fact we are currently extending the mechanism to provide separate `OMP_NUM_THREADS` environment variables for different parallel regions, i.e. provide a per-region default number of threads.

Acknowledgment

We acknowledge support of this work by the project "Dioni: Computing Infrastructure for Big-Data Processing and Analysis." (MIS No. 5047222) which is implemented under the Action "Reinforcement of the Research and Innovation Infrastructure", funded by the Operational Programme "Competitiveness, Entrepreneurship and Innovation" (NSRF 2014-2020) and co-financed by Greece and the European Union (European Regional Development Fund).

References

[1] OpenMP Architecture Review Board, "OpenMP application program interface version 5.2," Nov. 2021.

[2] T. Tzen and L. Ni, "Trapezoid self-scheduling: a practical scheduling scheme for parallel compilers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 1, pp. 87–98, Jan. 1993.

[3] E. Markatos and T. LeBlanc, "Using processor affinity in loop scheduling on shared-memory multiprocessors," *IEEE Transactions on Parallel and Dis-*

tributed Systems, vol. 5, no. 4, pp. 379–400, Apr. 1994.

[4] J. H. Müller Korndörfer, F. M. Ciorba, A. Yilmaz, C. Iwainsky, J. Doerfert, H. Finkel, V. Kale, and M. Klemm, "A runtime approach for dynamic load balancing of OpenMP parallel loops in LLVM," in *In Proc. SC19, International Conference for High Performance Computing, Networking, Storage, and Analysis*, Denver, CO, USA, 2019.

[5] V. V. Dimakopoulos, E. Leontiadis, and G. Tzoumas, "A portable C compiler for OpenMP v.2.0," in *Proc. EWOMP 2003, 5th European Workshop on OpenMP*, Aachen, Germany, Sept. 2003, pp. 5–11.

[6] C. Kruskal and A. Weiss, "Allocating independent subtasks on parallel processors," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 10, pp. 1001–1016, Oct. 1985.

[7] S. Lucco, "A dynamic scheduling method for irregular parallel programs," *SIGPLAN Not.*, vol. 27, no. 7, pp. 200–211, July 1992.

[8] S. F. Hummel, E. Schonberg, and L. E. Flynn, "Factoring: A method for scheduling parallel loops," *Commun. ACM*, vol. 35, no. 8, pp. 90—-101, Aug. 1992.

[9] S. F. Hummel, J. Schmidt, R. N. Uma, and J. Wein, "Load-sharing in heterogeneous systems via weighted factoring," in *Proc. SPAA 1996, 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, Padua, Italy, 1996, pp. 318–328.

[10] I. Banicescu, "Load balancing and data locality in the parallelization of the fast multipole algorithm," Ph.D. dissertation, Polytechnic University, Brooklyn, New York, 1996, uMI Order No. GAX96-12331.

[11] J. D. Booth and P. A. Lane, "An adaptive self-scheduling loop scheduler," *Concurrency and Computation: Practice and Experience*, vol. 34, no. 6, 2022.

[12] E. Ayguadé, B. Blainey, A. Duran, J. Labarta, F. Martínez, X. Martorell, and R. Silvera, "Is the schedule clause really necessary in OpenMP?" in *Proc. WOMPAT 2003, International Workshop on OpenMP Applications and Tools*, Toronto, Canada, June 2003, pp. 147–159.

[13] Y. Zhang, M. Burcea, V. Cheng, R. Ho, and M. Voss, "An adaptive OpenMP loop scheduler for hyper-threaded smps." in *Proc. ISCA 2004, 17th International Conference on Parallel and Distributed Computing Systems*, vol. 4, San Francisco, California, USA, 2004, pp. 256–263.

[14] P. Thoman, H. Jordan, S. Pellegrini, and T. Fahringer, "Automatic OpenMP loop scheduling: a combined compiler and runtime approach," in *Proc. IWOMP 2012, 8th International Workshop on OpenMP*, Rome, Italy, June 2012, pp. 88–101.

[15] F. Kasielke, R. Tschüter, C. Iwainsky, M. Velten, F. M. Ciorba, and I. Banicescu, "Exploring loop scheduling enhancements in OpenMP: An LLVM case study," in *Proc. ISPDC 2019, 18th International Symposium on Parallel and Distributed Computing*, Amsterdam, Netherlands, June 2019, pp. 131–138.

[16] F. M. Ciorba, C. Iwainsky, and P. Buder, "OpenMP loop scheduling revisited: Making a case for more schedules," in *Proc. IWOMP 2018, 14th International Workshop on OpenMP*, Barcelona, Spain, Sept. 2018, pp. 21–36.

[17] V. Kale, C. Iwainsky, M. Klemm, J. H. Müller Korndörfer, and F. M. Ciorba, "Toward a standard interface for user-defined scheduling in OpenMP," in *Proc. IWOMP 2019, 15th International Workshop on OpenMP*, Auckland, New Zealand, Sept. 2019, pp. 186–200.

[18] J. M. Bull, "Measuring synchronisation and scheduling overheads in OpenMP," in *Proc. of 1st European Workshop on OpenMP*, vol. 8, Lund, Sweden, 1999, p. 49.

[19] A. J. Dorta, C. Rodriguez, and F. de Sande, "The OpenMP source code repository," in *PDP2005, 13th Euromicro Conference on Parallel, Distributed and Network-Based Processing*. Lugano, Switzerland: IEEE, Feb. 2005, pp. 244–250.

[20] Department of Energy, "CORAL procurement benchmarks." LLNL-PRE-637694, May 2013.

[21] W. Swope, H. Andersen, P. Berens, and K. Wilson, "A computer-simulation method for the calculation of equilibrium-constants for the formation of physical clusters of molecules—application to small water clusters," *The Journal of Chemical Physics*, vol. 76, pp. 637–649, Jan. 1982.

[22] P. Romano, B. Herman, N. Horelik, A. Nelson, B. Forget, and K. Smith, "OpenMC: A state-of-the-art monte carlo code for research and development," *Joint International Conference on Supercomputing in Nuclear Applications and Monte Carlo 2013*, vol. 82, Jan. 2013.

[23] D. H. Bailey, "FFTs in external of hierarchical memory," in *Proc. of the 1989 ACM/IEEE conference on Supercomputing*, Reno, NV, USA, 1989, pp. 234–242.