# ScalOMP: Analyzing the Scalability of OpenMP Applications

Anton Daumen[1,2](✉), Patrick Carribault[1], François Trahay[2], and Gaël Thomas[2]

[1] CEA, DAM, DIF, 91297 Arpajon, France
{anton.daumen.ocre,patrick.carribault}@cea.fr
[2] Télécom SudParis, Institut Polytechnique de Paris, Évry, France
{francois.trahay,gael.thomas}@telecom-sudparis.eu

**Abstract.** Achieving good scalability from parallel codes is becoming increasingly difficult due to the hardware becoming more and more complex. Performance tools help developers but their use is sometimes complicated and very iterative. In this paper we propose a simple methodology for assessing the scalability and for detecting performance problems in an OpenMP application. This methodology is implemented in a performance analysis tool named ScalOMP that relies on the capabilities of OMPT for analyzing OpenMP applications. ScalOMP reports the code regions with scalability issues and suggests optimization strategies for those issues. The evaluation shows that ScalOMP incurs low overhead and that its suggestions lead to significant performance improvement of several OpenMP applications.

**Keywords:** Performance tool · Scalability · OMPT

## 1 Introduction

The lifespan of simulation codes is several times longer than the lifespan of supercomputers. Thus, a single code will be used on multiple very different architectures, making the portability and the optimization of codes difficult. Furthermore computer architectures are more and more complex as their design has to become more intricate in order for performance to continue increasing. In their chase for better performance, developers rely on performance analysis tools to understand and analyze their code.

Many performance tools provide a wide range of features, metrics, and analysis. However the more features a performance tool has, the more complex it is to use. The developer has to learn how to use the tools in order to start efficiently using them for code analysis. Furthermore a lot of tools use an incremental methodology for analyzing codes, the tool first reports the global behavior of the code and the developer then focuses his analysis on important regions. The developer then tries to detect the issues in said regions by using other features of the tool and analyzing the source code directly, forming hypothesis and using

X. Fan et al. (Eds.): IWOMP 2019, LNCS 11718, pp. 36–49, 2019.
https://doi.org/10.1007/978-3-030-28596-8_3

the tool to verify them and quantify the importance of a performance problem before trying to fix it in the code.

In this paper we propose a simple methodology for analyzing the performance of a parallel application with a focus on the scalability of OpenMP applications. This approach is implemented in ScalOMP, that reports directly to the user the parallel regions where time is lost due to scalability issues and to automatically deduce the sources of these losses in order for the developer to directly know where time was lost and why. The whole process needs to be as closely related to the source code as possible in order for the developer to immediately understand where an issue resides. When possible, ScalOMP provide hints on how an issue may be solved.

The remainder of this paper is organized as follows: in Sect. 2 we present state-of-the-art tools that illustrate the typical methodology of performance tools. We detail our methodology in Sect. 3, and we describe ScalOMP internals in Sect. 4. We evaluate our approach in Sect. 5, and in Sect. 6 we conclude this paper.

## 2 Related Work

The performance tool landscape is filled with a significant number of tools that provide a very broad variety of features.

A lot of effort has been spent on tools that help the developper better visualize the behavior of an application [8,15]. These tools allow developper to precisely examine the application execution, but the analysis has to be done manually.

Automatic analysis relieve the developper from the analysis. Several works have focused on detecting the root causes of scalability issues in parallel applications. Most of these works are focused on MPI; For example, performance models can help finding weak scaling issues [6]. A backward replay of an execution trace can be used for identifying the root cause of wait-states in MPI applications [5].

Other work focuses on detecting and reporting problems in multi-threaded applications. For instance, imbalance issues in OpenMP parallel regions and worksharing constructs can affect the scalability of an application [18]. Running micro-benchmarks and building a compositional model can predict the performance of OpenMP applications on a given machine [16]. However, this approach is limited to memory-bound applications and only works on OpenMP applications using the *static* scheduler. In order to detect false-sharing, a recent work uses the OMPT API to instrument OpenMP constructs, and collects hardware counters at a fine granularity [9]. The collected data then train a classifier which can then spot false sharing in applications. Automated performance modeling can be used to examine the scalability of OpenMP runtime constructs [12], or to analyze the memory access patterns of OpenMP applications [4].

While all of these approaches are functional and allow a developper to identify issues, verifying every possible problem using different tools is time consuming. Moreover each approach has its own requirements and limitations which can make using these tools together difficult, and does not match our view on how the performance analysis of an application should work.

Some tools do integrate multiple analysis successfully. Intel VTune [17] provides an OpenMP *time gain* analysis that estimates the potential gain that could be obtained if various performance problems (lock contention, imbalance, scheduling overhead, etc.) were fixed. However Vtune *time gain* is lacking a scalability analysis, which means that if a performance issue is not detected, a code region may be wrongfully considered as having no issue. Finally, VTune uses profiling to measure the time spent in OpenMP constructs. While this limits the instrumentation overhead, it also affects the measurements precisions and lacks some insight that tracing may give.

## 3 Performance Analysis of OpenMP Applications

As described in Sect. 2, even with modern performance tools, most of the analysis remains the work of the developer and is done manually. Our work focuses on alleviating this burden as much as possible from the developper hands. This section presents our approach for assessing the scalability of a multithreaded application and how a performance analysis tool can provide developers with optimisation hints. We implement this approach in ScalOMP, whose implementation details are described in Sect. 4.

### 3.1 Methodology

There are multiple sources of performance problems in OpenMP applications, such as load imbalance, or lock contention. Once the problem is identified, the developper may improve the performance of the application in several ways. Some issues require code changes, while changing the execution settings may fix some other problems. In this section, we describe a methodology for detecting performance problems in OpenMP applications, and providing optimization suggestions to the developper.

As an input, the developper provides a compiled version of the application, along with the command line that runs it. The application is instrumented and runs while varying the number of threads in order to measure the scalability of each parallel region and to detect performance problems. As a result, ScalOMP computes the parallel efficiency of each parallel region (defined by the speedup multiplied by the initial number of threads divided by the current number of threads), and estimates the potential time gain for each parallel region. The output is the list of parallel regions sorted according to the potential time gain, and for each region, a set of optimization hints and their respective potential time gain.

Using this approach, a developper can focus on optimizing the most promising parallel regions of his application. Moreover, the optimization hints indicate how the performance could be improved.

## 3.2 Scalability Analysis on Parallel Regions

The performance analysis of an application starts with a scalability analysis which aims at identifying the OpenMP parallel regions that may be worth optimizing. ScalOMP does this by running the application multiple times while automatically varying the number of threads across a range given as input. For each run, ScalOMP measures the duration of each parallel region. As a result, ScalOMP computes the parallel efficiency of all the regions and estimates the time lost in each by comparing their efficiency to the expected behaviour of a perfectly scalable region [7].

Thus, ScalOMP identifies the parallel regions where most time is lost and never underestimate a parallel region's impact due to not detecting its issues or wrongly quantifying their effects. These parallel regions are good candidates for optimizations: the poor scalability of a region means that some performance issues affects its parallel efficiency; and the significant amount of time lost means there is good hope of gaining back time by improving the region's performance.

## 3.3 Quantifying the Impact of Performance Problems

A poor parallel efficiency in an OpenMP parallel region may be caused by several types of problems. In this section, we describe some of these problems and how a performance analysis tool can estimate their impact on performance. As a result, it is possible to quantify the potential time gain for each problem in each parallel region.

**Barriers.** One of the main synchronization mechanisms in OpenMP is the barrier that allows threads to wait for each other. This synchronization may be explicit (when using the `omp barrier` directive), or implicit (e.g., at the end of a parallel for loop). While barriers allow developers to ensure the correctness of parallel programs, they introduce synchronization points that may degrade the parallel efficiency.

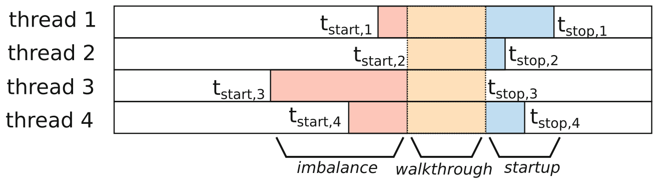

**Fig. 1.** Illustration of threads passing through an OpenMP barrier

As illustrated in Fig. 1, when a set of threads pass through an OpenMP barrier, three phases can be distinguished:

- the *imbalance phase* starts when the first thread enter the barrier, and ends when the last thread reaches the barrier. If a thread arrives late to a barrier, it delays the other threads. This means that the ideal case is when all the threads reach the barrier simultaneously. Hence the time lost by imbalance is the difference between the average time a thread took to reach the barrier from the last point of synchronization, and the maximum time. A long *imbalance phase* may be caused by an uneven work distribution between the threads, or by some delay that applies to a thread (such as a late MPI communication). If the imbalance is significant, ScalOMP suggests to improve the work distribution, for example by using a `schedule dynamic` clause in a parallel loop.
- the *walkthrough phase* starts when all the threads have reached the barrier and ends when the first thread leaves the barrier. We consider this phase to be the incompressible time spent resolving the barrier. This is an optimistic estimation since part of the barrier is resolved every time a thread arrives. If the walkthrough phase takes a significant part of the overall execution time, ScalOMP suggests to either reduce the number of OpenMP barriers, or to improve the barrier algorithm (for instance, by choosing a more performant OpenMP runtime)
- the *startup phase* starts when the first thread leaves the barrier and ends when the last thread leaves the barrier. The time lost from threads not leaving at the same time is paid at the next point of synchronization. For example if there is no imbalance in the work of threads between two barriers, some tools could detect differences in arrival times and report it as imbalance when the real culprit is the previous barrier delay when releasing threads. ScalOMP detects those delays and shifts the blame to the previous barrier runtime instead.

**Locks.** Locking is another major synchronization mechanism in OpenMP that may significantly impact the performance. As depicted in Fig. 2, the time spent acquiring a lock can be separated in two phases:

- the *waiting* phase happens when a thread tries to acquire a lock that is currently held by another thread. This phase corresponds to the contention that applies to the lock. To reduce the waiting phase, ScalOMP suggests to either change the application to reduce the number of concurrent access to this lock, or to use another locking mechanism that is less affected by contention (for instance MCS or AHMCS [10]).
- the *acquisition* phase happens when the lock is available. This phase corresponds to the incompressible time required for running the locking algorithm. A significant part of the whole execution spent in the acquisition phase means that the thread often acquires locks without contention. In that case, ScalOMP suggests to either reduce the number of calls to locking primitives, or to use another synchronization mechanism (such as the `atomic` directive or a locking mechanism that works better with no contention).

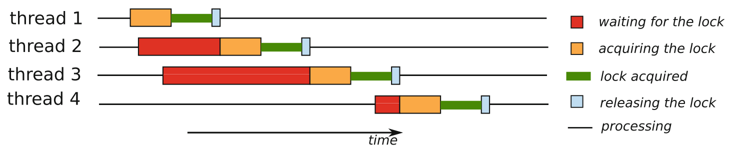

**Fig. 2.** Illustration of threads acquiring a lock

In order to estimate how much time is due to the contention, and to the lock algorithm, ScalOMP measures the time spent acquiring the lock and assumes that the fastest measured acquisition was contention free. This gives an estimate of the constant time required for executing the locking algorithm, and the remaining time is attributed to the contention on the lock.

## 4 Implementation

In order to compute the metrics described in Sect. 3, we implemented ScalOMP. In this section, we detail how ScalOMP instruments an OpenMP application, and how it measures the duration of OpenMP constructs without altering the application behavior.

### 4.1 Instrumenting an OpenMP Application with OMPT

As OpenMP relies on both compiler directives and a runtime API, instrumentation can be tedious. One solution consists in building a set of wrappers that intercept the calls to the OpenMP runtime API. However, this method is specific to one OpenMP implementation and it cannot grasp the whole OpenMP semantics. Opari [14] performs a source-to-source transformation on the application and inserts POMP calls in the source code. This makes this approach more portable, but it requires to recompile the application.

ScalOMP uses the OpenMP Tools interface (OMPT) that was introduced in the OpenMP 5.0 standard [11]. OMPT makes the OpenMP runtime collaborate with performance analysis tools: the tools register callbacks for OpenMP events, and the OpenMP runtime then triggers the callbacks when the corresponding events happen. With this approach, ScalOMP can collect performance data from any OpenMP application without recompiling it.

### 4.2 Identifying OpenMP Parallel Regions

An application may consist of tens of OpenMP parallel regions, some of them being invoked multiple times. Thus, ScalOMP needs to identify a parallel region in order to aggregate the performance data from multiple calls to it. When the application starts an OpenMP parallel region, the OMPT interface invokes ScalOMP through a callback and provides a pointer to the OpenMP call in the application binary. The first time ScalOMP encounter an unknown callsite, it uses *libbfd* to retrieve the line of code associated with this address.

### 4.3 Measuring Temporal Data

As described in Sect. 3, ScalOMP analysis of the OpenMP barriers requires to collect several OpenMP events: ScalOMP needs to know when a thread starts a region's work, enters at a barrier, exits a barrier and ends the region. The lock analysis also requires to collect information when a thread starts and stops acquiring a lock. For each of these events, ScalOMP records a timestamp using the TSC counter. These timestamps are then used for measuring various durations in the thread processing. The TSC counter allows ScalOMP to record timestamps at a low cost, but these timestamps cannot be compared accross threads running on different sockets. Thus, ScalOMP also records a system-wide timestamp using `clock_gettime` at the beginning of each region execution in order to compare different threads timestamps.

### 4.4 Mitigating Instrumentation Overhead with Adaptive Sampling

While recording a timestamp using the TSC counter is lightweight, this overhead may significantly alter the application's performance if timestamps are recorded too often. In order to reduce this overhead, ScalOMP uses a sampling mechanism. Since the OMPT interface allows to dynamically activate or de-activate callbacks, ScalOMP only collects performance data on certain executions of a parallel region. The idea is that while two executions of a region may not be exactly the same, their behaviour is essentially similar.

Depending on the parallel region, the sampling frequency should be selected carefully: if a parallel region is only repeated a few times, all its executions should be captured, whereas a region that runs many times should only be captured from time to time. Thus, ScalOMP uses an adaptive sampling where the first executions are all measured, and then as the region is repeated, ScalOMP de-activates the callbacks for some executions. The more a region is repeated, the more often ScalOMP de-activates the OMPT callbacks.

As a result, the rare regions are all captured, while frequent regions are sparsely captured, and the overhead of ScalOMP on the application execution remains low.

## 5 Experiments and Results

In this Section, we evaluate ScalOMP implementation and assess how the performance analysis can help the developper improve a parallel application. First, we evaluate the overhead of ScalOMP on 16 applications. Then, we evaluate how ScalOMP detects load imbalance problems, and lock contention issues.

For our evaluation, we use a machine equipped with two Intel Xeon Haswell E5-2698 v3 processors with 16 cores each (32 cores in total), and 128 GB of RAM. The machine runs Linux version 3.10, and the applications were compiled with Intel Compiler version 17.0.6. For OpenMP we use the open-source OpenMP runtime from Intel now maintained in LLVM. When compiling applications, we use the `-O3` optimization level.

We evaluate ScalOMP using several OpenMP applications:

- **Mandelbrot** is an application that computes the Mandelbrot set;
- **HydroMM** is an hydrodynamics mini-application;
- **Lulesh 2.0** is a mini-application that performs an hydrodynamics simulation [13];
- **BT**, **CG**, **DC**, **EP**, **FT**, **IS**, **LU**, **MG**, **UA** are kernels from the OpenMP NAS Parallel Benchmarks version 3.3.1 [3];
- **miniFE** is a Finite Element mini-application [2];
- **Snap** is a particle transport mini-application [2];
- **AMG** is a parallel algebraic multigrid solver for linear systems [1];
- **Pennant** is a mini-application for hydrodynamics [1].

## 5.1 Overhead of ScalOMP

To evaluate the overhead of ScalOMP, we run the 16 applications described in Sect. 5 with and without ScalOMP. For each application, the problem size is chosen so that the reference time (i.e. the execution time when running without ScalOMP) is between 10 and 100 s with a few exception to see how scale affects the overhead. Each measurement is repeated 5 times and we report the average execution time. Table 1 reports the execution time when running the application without ScalOMP, and the overhead when running with ScalOMP.

**Table 1.** Overhead induced by the tool

| Application | Mandelbrot | HydroMM | Lulesh2.0 | BT.B | CG.C | DC.A | EP.D | FT.C | IS.D |
|---|---|---|---|---|---|---|---|---|---|
| Reference time | 11.34 s | 13.63 s | 82.02 s | 10.35 s | 12.23 s | 16.21 s | 52.67 s | 11.67 s | 27.09 s |
| Overhead | 2.24% | 6.18% | 12.34% | 0.00% | 0.03% | −0.27% | 3.55% | −0.55% | −0.01% |
| Application | LU.C | LU.D | MG.D | UA.B | UA.D | miniFE | Snap | AMG | Pennant |
| Reference time | 36.53 s | 1352.66 s | 88.06 s | 12.61 s | 1161.65 s | 18.49 s | 80.24 s | 94 s | 41.32 s |
| Overhead | −0.17% | 0.14% | 0.12% | 13.78% | 4.04% | 2.24% | −0.64% | −1.03% | 2.33% |

The results show that ScalOMP has little impact on the performance of most applications. The overhead is higher for Lulesh (12.34%) because this application performs many small parallel regions; The observed overhead goes down to −4.31% if the size is increased to 120 (from 80) and the number of iteration lowered to 350 (from 1000) so that the time stay similar. UA.B also suffers from a significant overhead (13.78%) due to the heavy number of lock operations (8.1M locks per second per thread on average). When running UA with a larger problem size (class D), the application take locks less often and the ScalOMP's overhead is reduced to 4.04%. We conclude that ScalOMP does not significantly alter the application execution except in some extreme cases.

## 5.2 Detecting Imbalance Issues

ScalOMP reports that several of the applications evaluated in Sect. 5.1 suffer from load imbalance. In this section, we focus on two of these applications.

**Mandelbrot.** When running the **mandelbrot** application with 32 threads, ScalOMP reports that the parallel efficiency is only 42%. ScalOMP reports that the load imbalance between the threads in one parallel region is responsible for all of the lost time. It suggests to improve the load balancing for this parallel region, as it predicts a perfect load balance may save 5.37 s.

Based on this suggestion, we analyze the source code of this application. The incriminated parallel region computes the divergence of a set of complex numbers in the Mandelbrot set. The computation cost for each complex number depends on how fast it diverges. The default OpenMP scheduling policy assigns many numbers that diverge quickly to some threads while some other threads have to process many numbers that diverge slowly. As a result, some threads finish their loop iterations earlier than the other threads, leading to a load imbalance and a poor parallel efficiency. As suggested by ScalOMP, we change the scheduling policy for this parallel region to `dynamic`, and we observe that the application's execution time is reduced to 6.16 s. This means we gained 5.20 s which is close to the 5.37 predicted. When analyzing the parallel region with *dynamic* ScalOMP find the load balance to be 99.9% perfect.

**HydroMM.** When running HydroMM with 32 threads, ScalOMP reports that the parallel efficiency is only 51%, and reports that the load imbalance in one parallel region is responsible for 87% of the total lost time. ScalOMP suggests to improve the load balancing of this parallel region, it also predicts the performance of the application if this parallel region was perfectly load balanced.

Based on ScalOMP suggestion, we analyze the source code of HydroMM, and change the OpenMP scheduling policy to `dynamic` in order to improve the load balancing between the threads. Figure 3 reports the speedup measured for HydroMM when using the default scheduling policy (`static`) and when using the `dynamic` scheduling policy. It also reports the speedup predicted by ScalOMP. We observe that changing the scheduling policy significantly improves the application's performance. The results also show that up to 16 threads, the speedup obtained when applying ScalOMP suggestion is close to the predicted speedup.

For 32 threads, the predicted speedup is significantly overestimated. This may be due to memory effects being ignored by ScalOMP: up to 16 threads, all the threads execute on one socket of the machine, while when running 32 threads, the two sockets are used.

## 5.3 Detecting Locking Issues

In this section, we assess how ScalOMP detects locking issues using two applications. First, we evaluate how ScalOMP differentiates contended locks and non-contended locks using a micro-benchmark. Then, we present a case study on the UA kernel from the NAS Parallel Benchmarks. For both applications, we compare the optimization suggestions provided by ScalOMP with those obtained with Intel VTune [17].

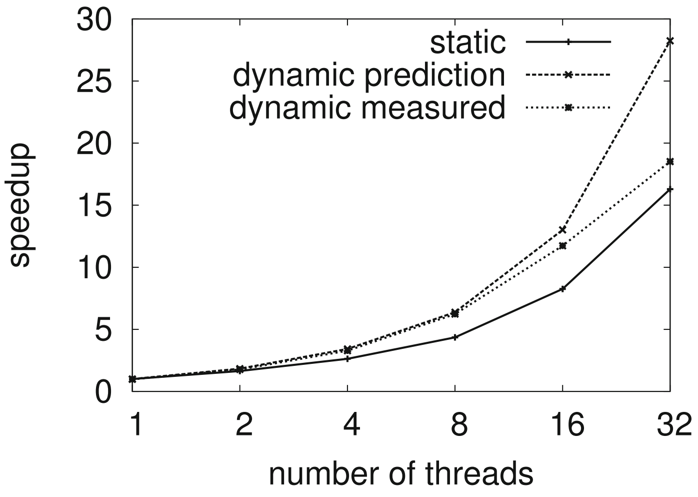

**Fig. 3.** Speedup obtained when running **HydroMM**

**Micro-benchmark.** We implemented an OpenMP application that consists in two parallel regions. In these parallel regions, each thread acquires an OpenMP lock, releases it, and busy waits for some time. In the first parallel region, each thread accesses a different lock which does not suffer from contention. In the second parallel region, all the threads access the same lock which suffers from contention. We choose the busy wait time so that the time spent acquiring lock is similar for both parallel region.

**Table 2.** Lock Micro-benchmark

| | | Without contention | With contention |
|---|---|---|---|
| | Total duration (s) | 14.14 | 14.34 |
| | Lock duration (s) | 4.63 (32%) | 4.35 (29.6%) |
| VTune | Overhead | 2.61% | 10.19% |
| | Lock contention | 2.64 s (18.19%) | 5.5 s (34.81%) |
| | Other problems | 0.67 s (4.6%) | 0.02 s (0.1%) |
| ScalOMP | Overhead | 4.00% | 1.04% |
| | Lock duration | 3.93 s (26.97%) | 5.24 s (36.45%) |
| | - Lock algorithm | 3.40 s (23.14%) | 0.03 s (0.23%) |
| | - Lock contention | 0.56 s (3.83%) | 5.21 s (36.22%) |

We run this micro-benchmark and analyze it with VTune and ScalOMP. The results of this experiment are reported in Table 2. The time spent acquiring locks and the total duration of the two regions are similar.

VTune detects significant lock contention in the contention-free region (18.19% of the region duration) and in the region with contention (34.81% of the region duration). VTune also reports that 4.6% of the time spent in region 1 is lost due to "Other" problems. We conclude that VTune is able to detect the lock contention problem in the second parallel region, but it wrongfully detects a lock contention in the first parallel region.

ScalOMP detects that a significant time is spent in locks in the contention-free region, and that most of it is due to the lock algorithm itself. ScalOMP indicates that the problem is that the threads acquire too many locks. In the second parallel region, ScalOMP detects that most of the locking time is due to contention. We conclude that ScalOMP rightly identifies the lock problems in the two parallel regions.

**Case Study: UA.** In this section, we analyze UA and apply optimizations based on the suggestions provided by ScalOMP. Since one of these optimizations gives an incorrect result with the Intel Compilers, we use the GNU Compilers version 7.3.0 in this section.

When running UA.B, ScalOMP measures a parallel efficiency of 56% with 32 threads. One parallel region is responsible for most of the time loss because of several problems: 10% of the total execution time is lost due to load imbalance; and 19.4% of the total execution time is spent acquiring locks in this parallel region.

ScalOMP also points that 2 704 354 500 locks are acquired during the 8.68 s execution of this parallel region, meaning that on average, each thread acquires a lock every 102 ns. Due to a high number of region execution (more than 2000), ScalOMP automatically uses the sampling mechanism described in Sect. 4.4 and records the duration of only 16% of the all locks acquisition. As a result, the overhead induced by ScalOMP remains low, but ScalOMP still captures a significant amount of performance data.

ScalOMP reports that most of the time spent acquiring locks is due to the lockings algorithm, and contention on locks is low in this application. An analysis of UA source code shows that there are 334 600 differents locks, which limits the probability of a thread acquiring a lock that is already taken by another thread. Moreover, most of the locks are used for protecting simple instructions such as `x = x + y`.

Based on this analysis, we create two additional implementations of the application:

- **UA-Hint** uses the lock hint mechanism from OpenMP 5.0 and specifies that the locks are uncontended. This allows the OpenMP runtime to select the lock implementation that performs the best when there is no contention;
- **UA-Atomic** replaces the critical sections protected by locks with OpenMP atomic operations when possible.

We compare the performance of these two implementation with the UA-Default implementation. Figure 4 reports the speedup of the implementations as

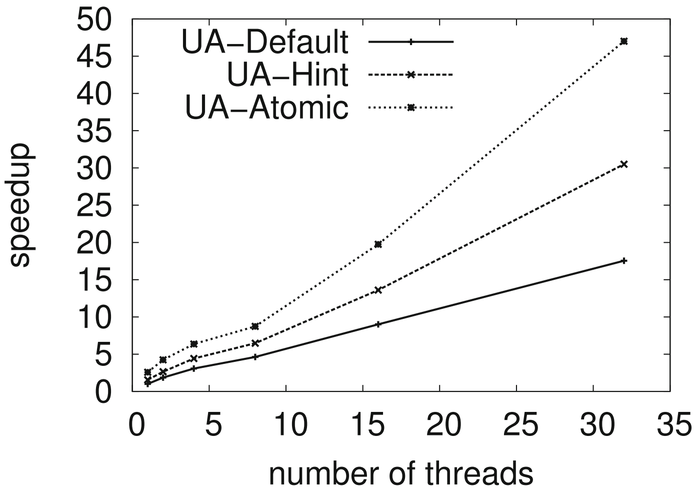

**Fig. 4.** Speedup obtained when running **UA**

compared to the execution time obtained when running UA-Default with one thread. As suggested by ScalOMP, UA-Hint performs better than UA-Default in all cases (including in sequential) because the locking algorithm achieves better performance when the lock is uncontended. UA-Atomic outperforms UA-Default (by 267 on 32 threads) and UA-Hint (by 154 on 32 threads) for all the tested number of threads. This is due to the single atomic instruction that replaces a call to a locking function and the critical section.

We conclude that ScalOMP suggests optimizations that may significantly improve the performance of an OpenMP application that suffers from locking problems.

## 6 Conclusion

Performance analysis of an application involves a lot of work from the developper. In this paper we presented a methodology that focuses on the scalability of an application in order to help the developer improve its performance. We implemented this approach in ScalOMP that relies on the OMPT API to instrument OpenMP applications. The evaluation show that ScalOMP collect performance data from applications with a low overhead. The experiments also show that ScalOMP analysis successfully detect load imbalance problems, and locking problems in several applications. The optimization hints provided by ScalOMP help the developer significantly improve the application's performance.

## References

1. Coral-2 benchmarks. Technical report, Lawrence Livermore National Lab. (LLNL), Livermore, CA, USA. https://asc.llnl.gov/coral-2-benchmarks/index.php
2. Coral benchmarks. Technical report, Lawrence Livermore National Lab. (LLNL), Livermore, CA, USA. https://asc.llnl.gov/CORAL-benchmarks/
3. NAS parallel benchmarks applications (NPB). Technical report, NASA Advanced Supercomputing Division. https://www.nas.nasa.gov/publications/npb.html
4. Barthou, D., Rubial, A.C., Jalby, W., Koliai, S., Valensi, C.: Performance tuning of x86 OpenMP codes with MAQAO. In: Müller, M., Resch, M., Schulz, A., Nagel, W. (eds.) TTools for High Performance Computing 2009, pp. 95–113. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11261-4_7
5. Bohme, D., Geimer, M., Wolf, F., Arnold, L.: Identifying the root causes of wait states in large-scale parallel applications. In: 2010 39th International Conference on Parallel Processing, pp. 90–100 (2010)
6. Calotoiu, A., Hoefler, T., Poke, M., Wolf, F.: Using automated performance modeling to find scalability bugs in complex codes. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, p. 45 (2013)
7. Coarfa, C., Mellor-Crummey, J.M., Froyd, N., Dotsenko, Y.: Scalability analysis of SPMD codes using expectations. In: Proceedings of the 21th Annual International Conference on Supercomputing, ICS 2007, Seattle, Washington, USA, 17–21 June 2007, pp. 13–22 (2007)
8. Coulomb, K., Degomme, A., Faverge, M., Trahay, F.: An open-source tool-chain for performance analysis. Tools High Perform. Comput. **2011**, 37–48 (2012)
9. Ghane, M., Malik, A.M., Chapman, B., Qawasmeh, A.: False sharing detection in OpenMP applications using OMPT API. In: International Workshop on OpenMP, pp. 102–114 (2015)
10. Guerraoui, R., Guiroux, H., Lachaize, R., Quéma, V., Trigonakis, V.: Lock-unlock: is that all? A pragmatic analysis of locking in software systems. ACM Trans. Comput. Syst. (TOCS) **36**(1), 1 (2019)
11. Huck, K.A., Malony, A.D., Shende, S., Jacobsen, D.W.: Integrated measurement for cross-platform OpenMP performance analysis. In: DeRose, L., de Supinski, B.R., Olivier, S.L., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2014. LNCS, vol. 8766, pp. 146–160. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11454-5_11
12. Iwainsky, C., et al.: How many threads will be too many? On the scalability of OpenMP implementations. In: Träff, J.L., Hunold, S., Versaci, F. (eds.) Euro-Par 2015. LNCS, vol. 9233, pp. 451–463. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48096-0_35
13. Karlin, I., Keasler, J., Neely, J.: LULESH 2.0 updates and changes. Technical report, Lawrence Livermore National Lab. (LLNL), Livermore, CA, USA (2013)
14. Knüpfer, A., et al.: Score-P: a joint performance measurement run-time infrastructure for periscope, Scalasca, Tau, and Vampir. In: Brunst, H., Müller, M., Nagel, W., Resch, M. (eds.) Tools for High Performance Computing 2011, pp. 79–91. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31476-6_7
15. Müller, M.S., et al.: Developing scalable applications with Vampir, Vampirserver and Vampirtrace. In: Parallel Computing (PARCO), vol. 15, pp. 637–644 (2007)

16. Putigny, B., Goglin, B., Barthou, D.: A benchmark-based performance model for memory-bound HPC applications. In: 2014 International Conference on High Performance Computing & Simulation (HPCS), pp. 943–950 (2014)
17. Reinders, J.: VTune performance analyzer essentials (2005)
18. Woodyard, M.: An experimental model to analyze OpenMP applications for system utilization. In: Chapman, B.M., Gropp, W.D., Kumaran, K., Müller, M.S. (eds.) IWOMP 2011. LNCS, vol. 6665, pp. 22–36. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21487-5_3