# Towards Unifying OpenMP Under the Task-Parallel Paradigm

## Implementation and Performance of the `taskloop` Construct

Artur Podobas(✉) and Sven Karlsson

Technical University of Denmark, Kongens Lyngby, Denmark
{podobas,svea}@dtu.dk

**Abstract.** OpenMP 4.5 introduced a task-parallel version of the classical thread-parallel for-loop construct: the taskloop construct. With this new construct, programmers are given the opportunity to choose between the two parallel paradigms to parallelize their for loops. However, it is unclear where and when the two approaches should be used when writing efficient parallel applications.

In this paper, we explore the taskloop construct. We study performance differences between traditional thread-parallel for loops and the new `taskloop` directive. We introduce an efficient implementation and compare our implementation to other taskloop implementations using micro- and kernel-benchmarks, as well as an application. We show that our taskloop implementation on average results in a 3.2 % increase in peak performance when compared against corresponding parallel-for loops.

## 1 Introduction

Task- and thread-parallelism are two different paradigms used to exploit parallel patterns in applications. Thread-parallelism stems from the conceptual abstraction of user-level threads as proxies for physical processors. These threads are made explicitly visible for the user. Task-parallelism is on the other hand fully oblivious of the physical layout of the system and programmers are instead encouraged to focus on exposing parallelism rather than mapping parallelism onto threads. Task-parallelism is argued to be more versatile than thread-level parallelism [3].

Up until version 3.0, OpenMP was a thread-parallel framework. However, with OpenMP 3.0, OpenMP added task-parallelism. Practitioners have since debated what paradigm to use in each case.

With OpenMP 4.5, a new task level construct has been added, the `taskloop` construct. This means that applications that previously used the `omp for` construct can today be re-written using `taskloop` directives. Hence, practically eliminating the need for thread-parallelism. This also means that programmers now need to take a decision which of the two paradigms to use.

N. Maruyama et al. (Eds.): IWOMP 2016, LNCS 9903, pp. 116–129, 2016.
DOI: 10.1007/978-3-319-45550-1_9

We argue that the task-parallel paradigm should be exclusively used. We also argue that a well-implemented `taskloop` can be used instead of the thread-parallel for loops in OpenMP.

The present paper is one step towards verifying our hypotheses. We do so by evaluating how well the recently added `taskloop` construct performs compared to the traditional `for` constructs. We also propose an efficient `taskloop` implementation. We show that using `taskloop`s can on average be 3.2 % faster than corresponding parallel for loops.

In short, our contributions are as follows:

- We reveal implementation details of our `taskloop` implementation, which includes a novel way of load balancing tasks.
- We evaluate the `taskloop` construct using our prototype implementation and the latest GCC implementation. We further compare against parallel-for implementations in GCC and Intel's OpenMP libraries.

For the remaining paper, we will use the term *task-loop* when we refer to the new `omp taskloop` directive and use *parallel-for* when we refer to the old `omp for` directive.

The paper is structured as followed. Section 2 surveys existing OpenMP `taskloop` implementations and Sect. 3 describes our `taskloop` implementation. Section 4 describes the experimental method with results in Sect. 5. Section 6 position our contributions against similar methods. Finally, we conclude the paper in Sect. 7.

## 2 Existing OpenMP Task-Loop Implementations

The OpenMP 4.5 `taskloop` decomposes a canonical for-loop into a set of unique tasks, where each task is not bound to any specific processing thread. The task-loop is compliant with the earlier `omp for`; the main difference is the lack of schedule- and reduction-clauses in the task-loop. Because both approaches behave in a similar way, programmers can now choose which of the two paradigms to be used in their applications.

Today, the OpenMP `taskloop` construct is supported in two frameworks: OmpSs [8] and GCC version 6.1. Both implementations use a single thread to spawn the entire loop, visually illustrated in Fig. 1:a. GCCs OpenMP implements the decomposition inside the runtime system while OmpSs does so statically through compiler transformations. Unless specified by the programmer, GCCs OpenMP runtime will decompose the iteration space evenly across the number of threads ($num_{task} = num_{threads}$) while OmpSs always requires the programmer to specify the chunk size for each loop.

We have observed three main limitations to OmpSs' and GCCs approaches. First, neither systems recursively decompose tasks, which also means that tasks are not decomposed in parallel. Instead, a single thread bears the responsibility of creating tasks. This leads to increased overheads and a longer critical path.

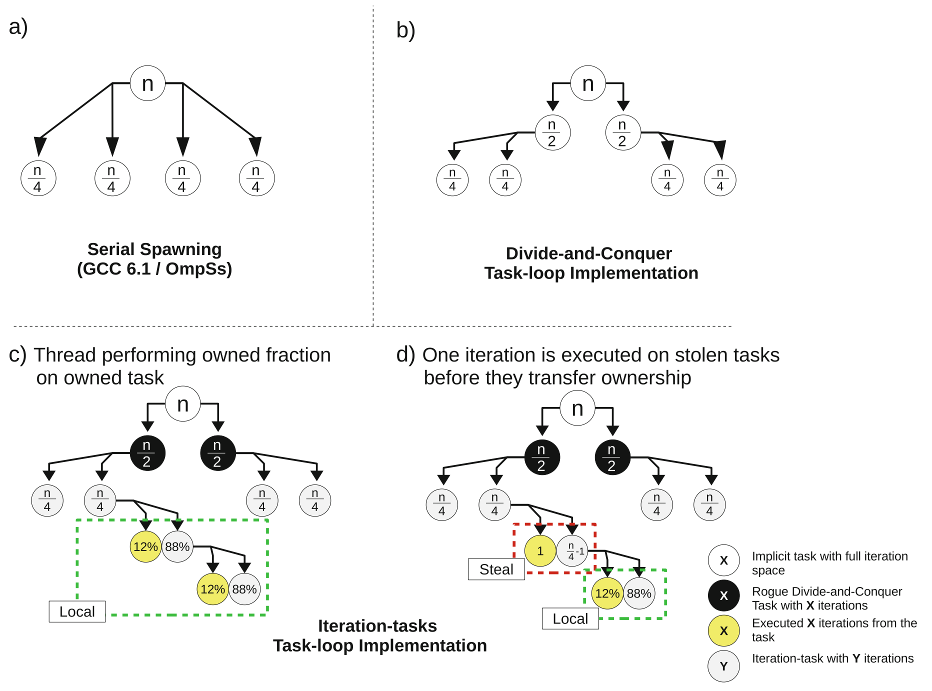

a)
Serial Spawning
(GCC 6.1 / OmpSs)
b)
Divide-and-Conquer
Task-loop Implementation
c) Thread performing owned fraction on owned task
d) One iteration is executed on stolen tasks before they transfer ownership
Local
Steal
Iteration-tasks
Task-loop Implementation
Implicit task with full iteration space
Rogue Divide-and-Conquer Task with X iterations
Executed X iterations from the task
Iteration-task with Y iterations

**Fig. 1.** Different task-loop decomposition strategies. (a) Serial spawn order found in GCC and OmpSs. (b) Divide-and-conquer found in runtime systems such as Cilk+ and Threading Building Blocks. (c)-(d) Proposed iteration tasks

Second, iterations are statically assigned with few opportunities for load balancing.

Third, and this only applies to GCC, a single global local is used when decomposing tasks, which increases the critical path.

## 3 Improved Task-Loop Implementation and Load Balancing

Based on our observations of the limitations in OmpSs and GCC, we have designed and implemented a scalable task-loop. We have included the implementation into Błysk.

Błysk[1] is our prototype runtime system, primarily developed for research on task-parallel runtime systems. Błysk is API compatible with GCCs OpenMP runtime system. Prior work on Błysk includes research on task dependencies and task-level speculation [6,14].

[1] The Błysk prototype implementation can be obtained through https://github.com/podobas/BLYSK.git.

### 3.1 Implementation

We have focused on the identified problems with GCCs OpenMP and OmpSs' implementation: overheads, task decomposition and load balancing. As with GCCs OpenMP, our implementation makes use of the existing tasking infrastructure in Błysk. The infrastructure provides functionality for maintaining and scheduling tasks. Each thread has a private task-queue, and load balancing is performed through work-stealing [9]. Accesses to shared data-structures are done through lock-free operations where possible, which is unlike the global locks used in GCCs OpenMP.

A task-loop in Błysk is decomposed using a recursive divide-and-conquer algorithm, illustrated in Fig. 1:b.

The benefits of recursively decomposing work through divide-and-conquer are many and range from improved cache utilization to a shorter critical path [5,11,13]. This is an improvement over GCCs OpenMP and OmpSs' approach where tasks are created serially by a single thread.

The divide-and-conquer algorithm will divide the iteration space into finer and finer tasks, until a cutoff is reached. The cutoff is similar to that of GCCs OpenMP runtime system. We subdivide the iteration space until we have as many tasks as we have threads. These leaf tasks of the task graph are called *iteration tasks*. We currently subdivide in a binary fashion but for large systems a higher branching factor is likely more efficient.

### 3.2 Iteration Tasks

Iteration tasks are the primary way to balance the workload across threads. An iteration task will have a subset of the iteration space associated with it.

Inspired by existing thread-parallel self-scheduling algorithms, we created a method for load balancing iteration tasks. The algorithm is visually illustrated in Fig. 1:c-d.

When a thread creates an iteration task, it will become the owner of that task and will execute iterations from its iteration space. However, there are situations where some threads have executed all their iterations while other threads have iterations left. To handle that, we allow threads to take over, or *steal*, iterations from other threads.

As an owner, a thread reserves and executes a large fraction of the iteration task's work, the *owned fraction*, seen in Fig. 1:c. We have heuristically chosen the owned fraction to be $\frac{1}{8}$ in this paper. The thread will steadily consume fractions of the owned task, decreasing the work in the task until all work is consumed. We allow threads to steal work from each other. Any unreserved amount of work in an iteration task can be stolen, which means that more than one thread can cooperatively consume iterations. In other words, when a thread tries to reserve work, another thread might already have stolen it.

When a thread has consumed the work within the task it owns, it will proceed to steal work from other threads, see Fig. 1:d. A successful steal will not only steal a single iteration from the task but also migrate the ownership to the thread that performed the steal. The thread first executes the stolen single iteration. It now owns the iteration task and will continue to reserve and consume iterations reserving an owned fraction in each step.

We have run a number of experiments with different configurations arriving at the aforementioned heuristic. We have explored scheduling and work stealing approaches, including stealing single iterations, not transferring ownership and various owned fractions. We found that an owned fraction of $\frac{1}{8}$ reaches equal performance to more fine-grained strategies but requires fewer steals. We present empirical evidence motivating our decision in Sect. 5.1.

## 4 Experimental Method

We used a 48 core AMD Opteron 6172-based system for the evaluation. The system consists of four sockets with two AMD Opteron 6172 processors per socket. Each processor contains 6 processing cores. The system runs at 2.1 GHz and have a total of 64 GB of RAM. The operating system running on the system is CentOS 6.5 with Linux kernel 2.6.32.

GCC version 6.1 was used to build all benchmarks as well as the Błysk runtime system. All benchmarks were compiled with aggressive optimizations, `-O3`. Intels OpenMP compiler version 15.0.3 was used to complement the performance evaluation.

### 4.1 Benchmarks

We evaluated our implementation as well as the GCC and Intels OpenMP implementations using benchmarks and microbenchmarks. It should be noted that our version of Intels OpenMP implementation does not yet support task-loops and so it is not used for any task-loop experiments.

We have developed two microbenchmarks. The first microbenchmark is a stress-test designed to evaluate the runtime systems' resilience to iteration granularity. The microbenchmark contains a for loop where iterations have a controllable amount of work. The chunk size is set to one to isolate per iteration overheads. We have two versions of the benchmark parallelized using task-loop and parallel-for respectively. We vary the amount of work in each iteration, the granularity, and measure the execution time.

The second microbenchmark tests the resilience to variance in the execution time of iterations. We use measurements from the first benchmark to design a benchmark where the average iteration granularity is large enough so that we see linear scalability with all the runtime systems. However, we also introduce a controlled uniformly random variance to the execution time of iterations. We make several runs with different levels of variance to observe the load balancing abilities of the runtime systems.

We have also used kernels from the SpecOMP benchmark suite [2], the Rodinia benchmark suite [7] and Parsec [4]. Table 1 shows the input data sets for the different benchmarks as well as serial execution time. We have selected

**Table 1.** Benchmarks, their input data set and serial execution time

| Benchmark | Input | Source | Serial time |
|---|---|---|---|
| 358.botsalgn | prot.100.aa | SpecOMP 2012 | $22,25\,s$ |
| 359.botspar | $8000 \times 8000$ | SpecOMP 2012 | $88,83\,s$ |
| Backward Propagation | 4194304 nodes | Rodinia | $3,87\,s$ |
| HeartWall | test.avi / 50 frames | Rodinia | $107,46\,s$ |
| HotSpot3D | $512 \times 8$ | Rodinia | $4,58\,s$ |
| LavaMD | 10 boxes | Rodinia | $65,81\,s$ |
| Leukocyte | testfile.avi / 10 frames | Rodinia | $71,20\,s$ |
| Prime | all primes 0-300,000 | Selfmade | $77,84\,s$ |
| BlackScholes | in_10M.txt | Parsec | $281,68\,s$ |

benchmarks which GCC 6.1 can handle. GCC 6.1 is, as of this writing, the current GCC version which also has some shortcomings.

GCC can currently, for example, not handle the case when a `parallel` or `single` construct is encountered in the same compound statement as a `taskloop` construct.

We compare the performance when using GCCs task-loop implementation as well as our own. In addition, we also measure the performance of benchmarks when using parallel-for.

The main metric for performance chosen is speed-up over the serial implementation, given as: $\frac{t_{serial}}{t_{parallel}}$

The speed-up shows how well the performance of the runtime system and application scales with the number of processors. All benchmarks were executed ten times and we show the mean value.

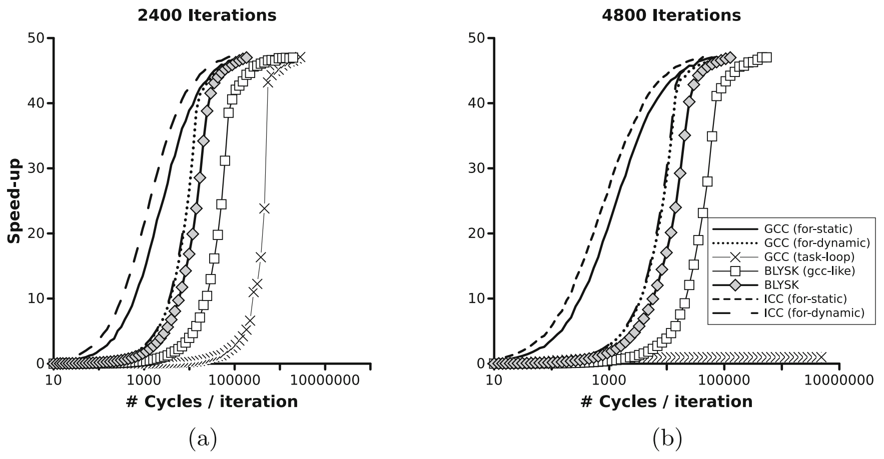
## 5 Results

We study performance in two principle ways. First, we study the scalability and load balancing capabilities of different task-loop and parallel-for implementations. Second, we observe the execution time of a set of benchmarks.

We use the two microbenchmarks we developed to study how well task-loop and parallel-for implementations scale and how well they handle load balancing problems.

First, we use our first microbenchmark and vary the number of loop iterations and their execution time. This will show what overheads runtime system incur with different number of iterations and iterations with different execution times. Figure 2 shows results for 2400 and 4800 iterations. The speed-up over a serial loop execution is plotted on the y-axis against the execution time of each iteration.

The best performing implementation is that of the older statically scheduled parallel-for loops, where Intel's OpenMP implementation performs with as low as 24 cycle iteration granularity and GCC's OpenMP implementation requires



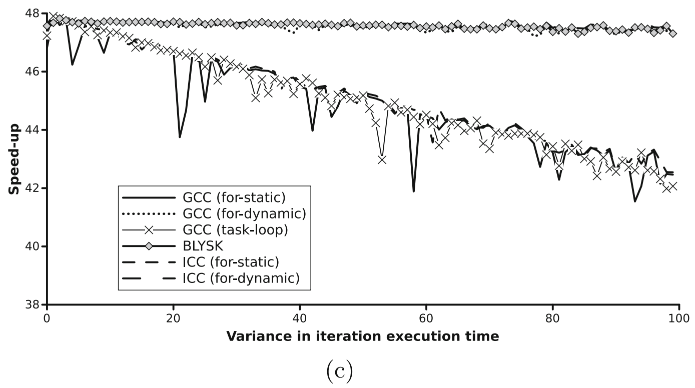


**Fig. 2.** Evaluation of different scheduling strategies using microbenchmarks with respect to: (a–b) resilience to iteration granularity, and (c) resilience to variance in iteration execution time

400 cycles per iteration. Both Intel's and GCC's parallel-for with a dynamic schedule perform worse than their static counterpart. GCC's task-loop implementation, GCC task-loop in Fig. 2, has the poorest performance, requiring coarser than 150,000 cycles/iteration to start performing well. Our re-implementation of the same algorithm, labeled "BLYSK (gcc-like)", results in better performance and start scaling at nearly 148,000 cycles shorter iterations. The increased performance of the Błysk re-implementation is due to Błysks better tasking infrastructure. For our proposed improved implementation, called Błysk in Fig. 2 and other figures, the Błysk scheduler can reach performance levels comparable to those of Intel's or GCC's dynamic parallel-for. Note how GCC's task-loop implementation refuses to scale in the case of 4800 iterations. Here GCCs runtime system actually refuses to spawn parallel tasks. GCC will not create any tasks if the aggregated number of spawned tasks is higher than

64 times the number of executing threads. That is the case in this situation and so the task-loop is executed serially.

We now use the second microbenchmark. We use a granularity of 1 million cycles per iteration to allow all runtime systems to scale to 48 threads. Figure 2:c shows the results of the second microbenchmark, where we see the speed-up plotted against the variance in per iteration execution time. Here the static parallel-for scheduler is showing its weakness. As we increase the variance, the load balancing properties of the static schedule decreases, yielding lower and lower performance. Because the algorithm of parallel-for static and GCC's OpenMP task-loop are similar, they both suffer from the same weakness to increased variance.

Intel's and GCC's parallel-for with dynamic schedule and our implementation of `taskloop` continue to scale to 48 threads, unaffected by the variance in iteration granularity.

## 5.1 Scheduling Heuristics

We now argue for our chosen scheduling heuristic, where we set the owned fraction to 12 % ($\frac{1}{8}$).

Figure 3 shows results of the second microbenchmark when executed under different scheduling heuristics with three different metrics in mind: performance, stealing activity and unbroken iterations. Executing long, unbroken iterations is important as it often honors the user's effort to write loops with good spatial and temporal cache locality [1]. A large amount of steals can lead to resource contention.

The evaluated heuristics range from a single iteration up-to consuming half the remaining iterations of a task.

Figure 3:a shows the performance when varying the amount of iterations consumed from tasks. Performance vary with the owned fraction, with diminishing returns as the owned fraction is reduced. For high performance, we need to use an owned fraction no larger than 12 %!

The number of steals that are occurring using the different heuristics is shown in Fig. 3:b. A steal is recorded every time a thread has to take iterations from a task it does not own. We see that consuming a smaller owned fraction leads to increased stealing activity. For example, using a single iteration leads to the highest stealing activity while using an owned fraction of 50 % results in the least amount of steals. This is intuitive as consuming a larger owned fraction leaves fewer iterations for potential load balancing purposes. The 12 % heuristic we have chosen steals on average 25 % less than the 6 % heuristic and 40 % less than the single iteration case.

Figure 3:c shows the average length of unbroken iterations for the three heuristics that yield the best performance. Our heuristic executes the longest continuous chain of iterations.

To summarize, our results shows that stealing 12 % of the iterations of a task in a controlled benchmark offers good performance with the fewest number of steals and the longest unbroken iteration chain.

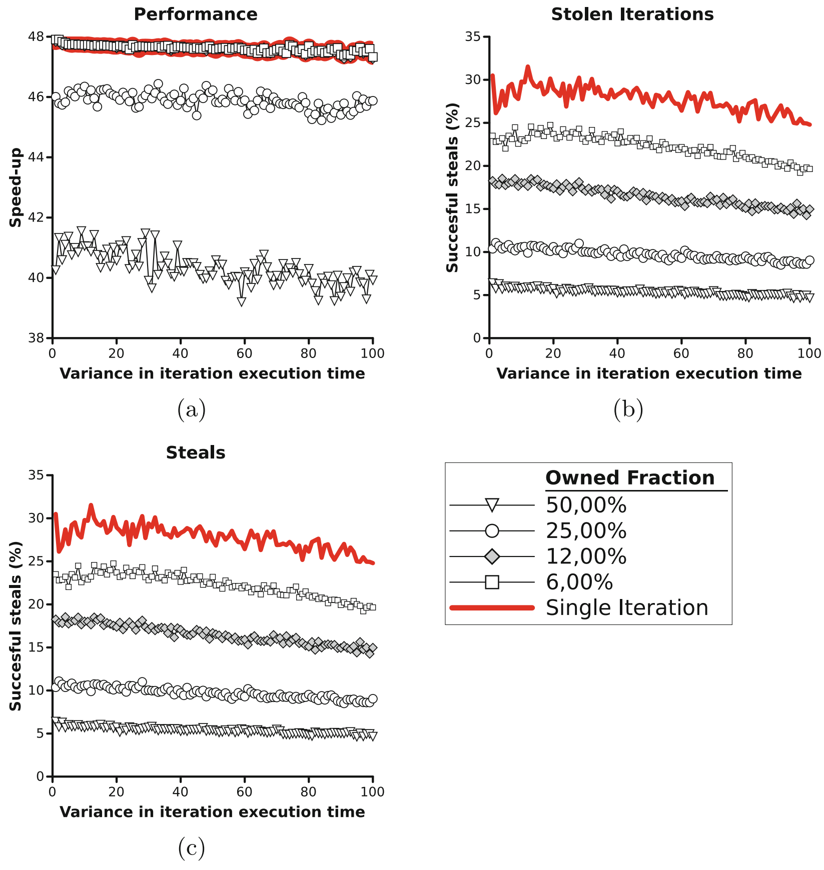


**Fig. 3.** Evaluation of different scheduling heuristics with respect to (a) performance, (b) number of steals and (c) average unbroken iteration length

### 5.2 Benchmark Performance

Figure 4 shows the performance of the various benchmarks. Here the speed-up is plotted against the number of threads allocated to the application.

For most benchmarks, the performance of the parallel-for version is comparable with our task-loop implementation. Some benchmarks, such as `rodinia.Backprop`, `rodinia.Hotspot` and `rodinia.Heartwall` scale poorly. This is mainly due to the lack of enough parallelism to occupy the 48 cores in the system. Others, such as `rodinia.lavaMD`, show an increase in performance for our proposed task-loop version when compared against GCC's parallel-for version. `rodinia.Leokocyte` and `parsec.BlackScholes` are both fairly coarse-grained applications that scale well for all runtime systems irrespective of the chosen paradigm.

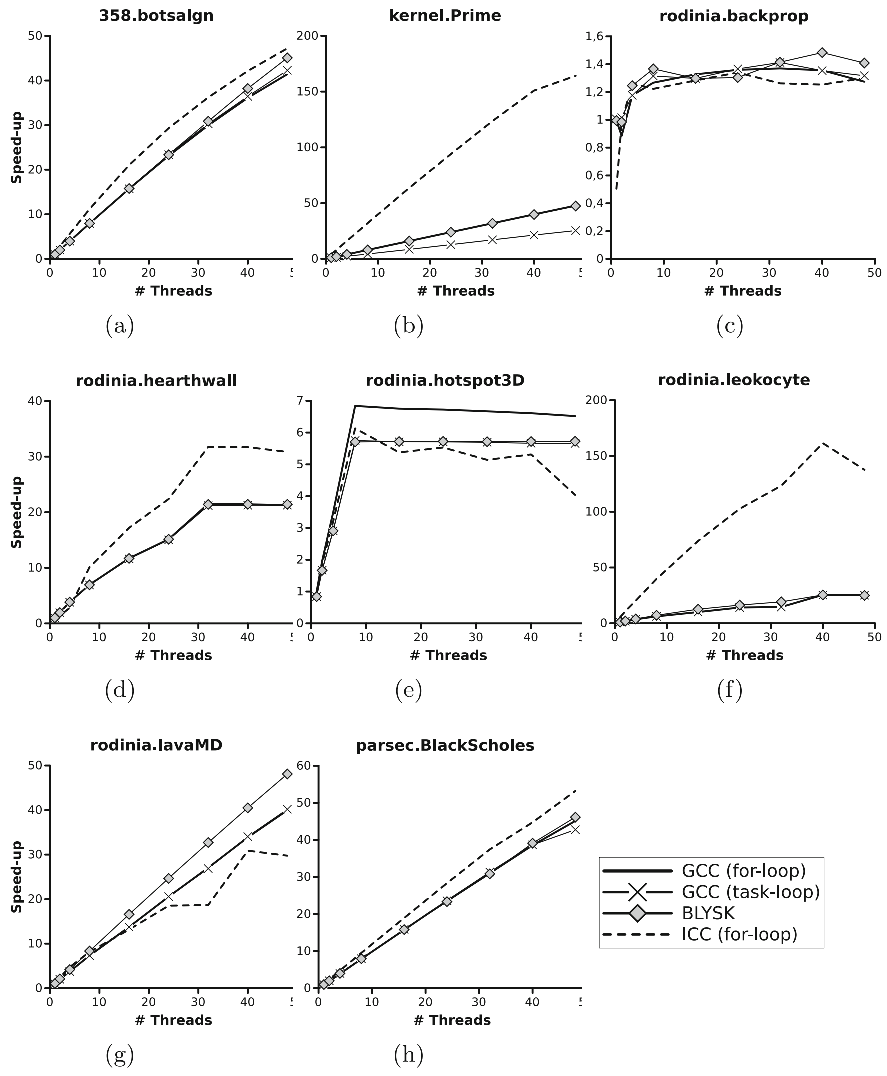


**Fig. 4.** Speed-up of benchmarks using task-loop and parallel-for OpenMP constructs

The `358.botsalign` benchmark shows a small performance difference between our and GCCs OpenMP implementations, to our advantage. The performance increase is mainly due to the hybrid nature of the application. Unlike other benchmarks, `358.botsalign` exploits both task- and thread-parallelism. The performance improvements of our task-loop come from the faster tasking infrastructure compared to GCCs OpenMP. However, note that GCCs implementation of task-loop slightly outperforms the GCC parallel-for version on `358.botsalign`, indicating that mixing the two paradigms is unfavorable.

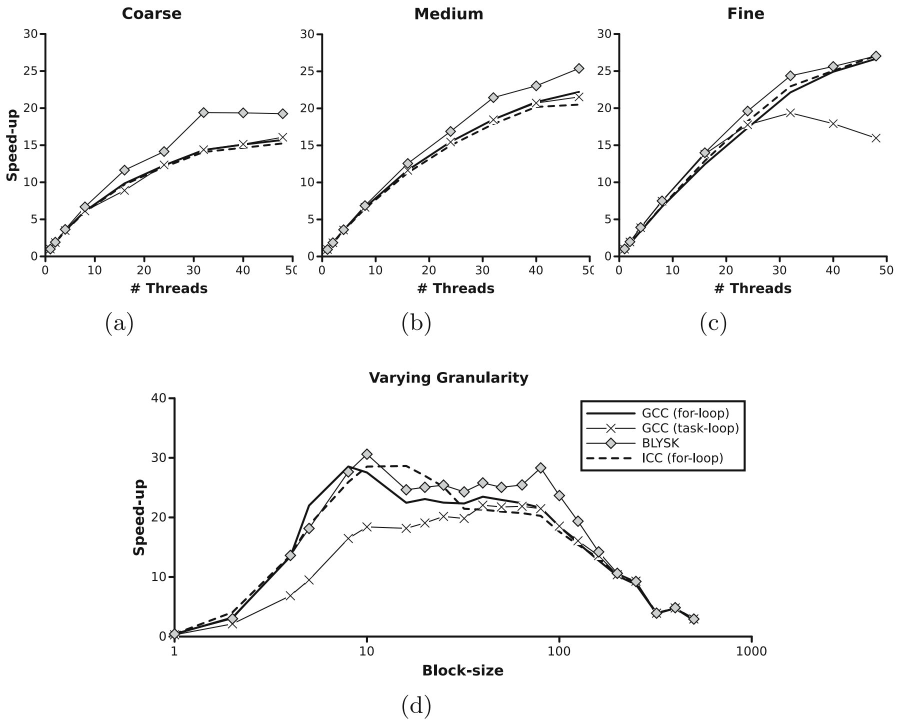


**Fig. 5.** Performance of the sparse LU benchmark when varying the task granularity.

Overall, our implementation of the task-loop outperforms the GCC task-loop implementation in all cases. This is attribute to both lower overheads and better load balancing.

The case of overheads is more clearly shown in Fig. 5:a–c. The figure shows `359.botspar` which implements a sparse LU factorization based on either the parallel-for or task-loop implementation. Here we see the performance obtained when the entire matrix is split into block-sizes of varying granularity. The various granularities represent the amount of parallelism available in the application to solve the LU decomposition.

Our task-loop implementation outperforms the other runtime systems consistently. For the coarse-grained case, the main reason is the poor decomposition done by the other runtime systems, which can be fixed by manually tuning grain-sizes. Notice how GCCs task-loop implementation fails to scale in the fine-grained case. This decrease in performance is due to the overhead increase associated with tasking. We also see that the overall performance increase as we decrease the granularity, which leads to increased parallelism and thus better performance.

Figure 5:d shows the speed-up as a function of the amount of parallelism, when varied from decomposing the matrix into $1 \times 1$ sized blocks all the way

up to $500 \times 500$ sized blocks. Note how the highest performance is reached by our task-loop implementation at a block-size of $8 \times 8$. The valley between the two peak points is mainly due to the uneven balance of parallelism that these block-sizes yield. While the parallelism increases, it does not increase enough to actually reduce the critical path of the application. The amount of application-level parallelism cannot be evenly distributed across the threads, which leads to some threads being idle.

We have included the performance of Intel's OpenMP implementation primary as a source of reference. However, it is more complicated to scrutinize the performance because the compilation infrastructure is different between Intels compiler and GCC. For example, the `kernel.Prime` benchmark performance is in part due to the vectorization capabilities of the Intel compiler. On the other hand, Intels runtime system degrades in performance on benchmarks such as `rodinia.Hotspot3D` and `rodinia.lavaMD`. We have not attempted to isolate the performance losses in Intel's OpenMP runtime system.

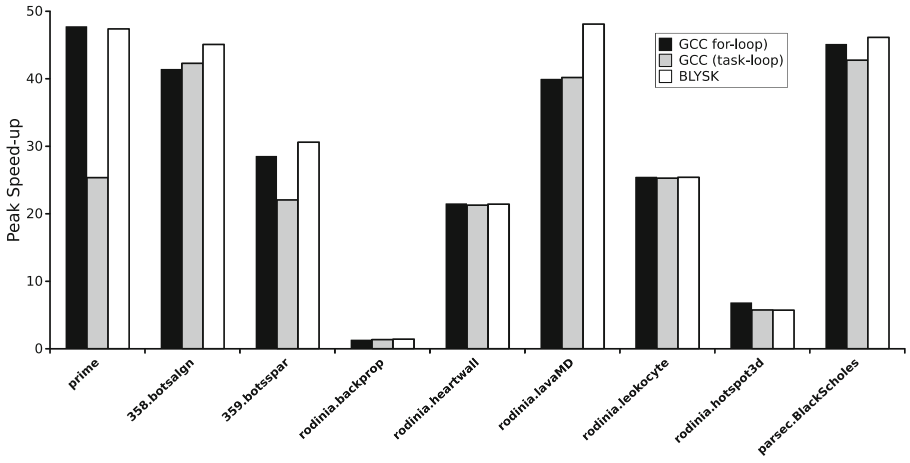


**Fig. 6.** The peak speed-up observed running with the different implementations for all the benchmarks using GCC's compilation framework

Figure 6 shows a summary of the peak performance that was obtained for each of benchmarks under the different implementations. The average reduction in performance when migrating from GCCs parallel-for to GCCs task-loops amount to a decrease of 9.6 % in performance. The average difference in performance between GCCs parallel-for and our improved task-loop is on the other hand an *increase* of 3.2 % in peak performance.

## 6 Related Work

Task based scheduling of for loops exists in several well-known task-based frameworks, such as Cilk++/Intel Cilk+ [12] and Intel Threading Build Blocks [10].

These frameworks decompose the iteration space using divide-and-conquer algorithms, most commonly using a binary division such similar to the one used in our implementation of task-loop. However, unlike our implementation, which introduces a new type of tasking primitive, the iteration tasks, said frameworks spawn a large number of tasks. Our approach is to spawn no more tasks than the number of threads and still achieves good load balancing properties.

Our task-loop approach is more closely related to that of classical thread-parallel iteration scheduling policies, though we adapt it to the task-parallel paradigm. More precisely, we position our work closely to the Chunk or Guided Chunk Self-Schedulers, GSS [15]. The main differences between our method and GSS is the parallelization paradigm and that our approach use the concept of ownership to help balance iteration. In GSS the stolen iteration are fixed. Other similar work is the Trapzeoid Self-Scheduler [16] which is similar to GSS but with a linear rather than nonlinear decreasing iteration distribution. Our proposed algorithm is of the nonlinear kind.

Another direction is to use several different scheduling algorithms combined. These are called N-level schedulers where different schedulers are used on various invocations of the same for loop [18] or on subsets of the iteration space [17]. However, N-level schedulers require profiling which has been shown to be costly in the task-parallel paradigm.

## 7 Conclusions

Our main purpose for this paper is to evaluate the `taskloop` construct. To do so, we have introduced an efficient implementation for load balancing task-loop iterations. We have evaluated this and existing `taskloop` implementations and can show that using `taskloop` can on average be 3.2 % faster than corresponding parallel for loops.

We used established kernel- and application-benchmarks to evaluate performance and conclude that the performance of a task-loop implementation could rival that of a traditional thread-parallel for loop.

Based on our results, we argue that the task-parallel paradigm in OpenMP is now poised to displace the thread-parallel paradigm.

**Acknowledgments.** We acknowledge the reviewers for their suggestions in making this paper better. The research leading to these results has received funding from the ARTEMIS Joint Undertaking under grant agreement number 332913 for project COPCAMS.

## References

1. Acar, U.A., Blelloch, G.E., Blumofe, R.D.: The data locality of work stealing. In: Proceedings of the Annual ACM Symposium on Parallel Algorithms and Architectures, pp. 1–12. ACM (2000)

2. Aslot, V., Domeika, M., Eigenmann, R., Gaertner, G., Jones, W.B., Parady, B.: SPEComp: a new benchmark suite for measuring parallel computer performance. In: Eigenmann, R., Voss, M.J. (eds.) WOMPAT 2001. LNCS, vol. 2104, pp. 1–10. Springer, Heidelberg (2001)
3. Ayguadé, E., Copty, N., Duran, A., Hoeflinger, J., Lin, Y., Massaioli, F., Teruel, X., Unnikrishnan, P., Zhang, G.: The design of OpenMP tasks. IEEE Trans. Parallel Distrib. Syst. **20**(3), 404–418 (2009)
4. Bienia, C., Li, K.: PARSEC 2.0: a new benchmark suite for chip-multiprocessors. In: Proceedings of the Annual Workshop on Modeling, Benchmarking and Simulation, vol. 2011 (2009)
5. Bohme, D., Wolf, F., Supinski, D., Bronis, R., Schulz, M., Geimer, M.: Scalable critical-path based performance analysis. In: Proceedings of Parallel & Distributed Processing Symposium, pp. 1330–1340. IEEE (2012)
6. Bonnichsen, L., Podobas, A.: Using transactional memory to avoid blocking in OpenMP synchronization directives. In: Terboven, C., et al. (eds.) IWOMP 2015. LNCS, vol. 9342, pp. 149–161. Springer, Heidelberg (2015). doi:10.1007/978-3-319-24595-9_11
7. Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J.W., Lee, S.-H., Skadron, K.: Rodinia: a benchmark suite for heterogeneous computing. In: Proceedings of IEEE International Symposium on Workload Characterization, pp. 44–54. IEEE (2009)
8. Duran, A., Ayguadé, E., Badia, R.M., Labarta, J., Martinell, L., Martorell, X., Planas, J.: OmpSs: a proposal for programming heterogeneous multi-core architectures. Parallel Process. Lett. **21**(02), 173–193 (2011)
9. Goldstein, S.C., Schauser, K.E., Culler, D.E.: Lazy threads: implementing a fast parallel call. J. Parallel Distrib. Comput. **37**(1), 5–20 (1996)
10. González, C.H., Fraguela, B.B.: A generic algorithm template for divide-and-conquer in multicore systems. In: Proceedings of IEEE International Conference on High Performance Computing and Communications, pp. 79–88. IEEE (2010)
11. Kumar, P.: Cache oblivious algorithms. In: Petreschi, R., Persiano, G., Silvestri, R. (eds.) CIAC 2003. LNCS, vol. 2653, pp. 193–212. Springer, Heidelberg (2003)
12. Leiserson, C.E.: The Cilk++ concurrency platform. J. Supercomput. **51**(3), 244–257 (2010)
13. Mohr, E., Kranz, D.A., Halstead Jr., R.H.: Lazy task creation: a technique for increasing the granularity of parallel programs. IEEE Trans. Parallel Distrib. Syst. **2**(3), 264–280 (1991)
14. Podobas, A., Brorsson, M., Vlassov, V.: TurboBŁYSK: scheduling for improved data-driven task performance with fast dependency resolution. In: DeRose, L., Supinski, B.R., Olivier, S.L., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2014. LNCS, vol. 8766, pp. 45–57. Springer, Heidelberg (2014)
15. Polychronopoulos, C.D., Kuck, D.J.: Guided self-scheduling: a practical scheduling scheme for parallel supercomputers. IEEE Trans. Comput. **100**(12), 1425–1439 (1987)
16. Tzen, H.T., Ni, L.M.: Trapezoid self-scheduling: a practical scheduling scheme for parallel compilers. IEEE Trans. Parallel Distrib. Syst. **4**(1), 87–98 (1993)
17. Zhang, Y., Burcea, M., Cheng, V., Ho, R., Voss, M.: An adaptive OpenMP loop scheduler for hyperthreaded SMPs. In: Proceedings of International Conference on Parallel and Distributed Computing (and Communications) Systems, pp. 256–263 (2004)
18. Zhang, Y., Voss, M., Rogers, E.S.: Runtime empirical selection of loop schedulers on hyperthreaded smps. In: Proceedings of International Parallel and Distributed Processing Symposium, p. 44b. IEEE (2005)