SAND2022-9631C

Characterizing the Performance of Task Reductions in OpenMP 5.X Implementations *

Jan Ciesko[0000−0003−3148−4477] and Stephen L. Olivier[0000−0001−6247−8980]

Center for Computing Research, Sandia National Laboratories
Albuquerque NM 87123, USA
{jciesko, slolivi}@sandia.gov

Abstract. OpenMP 5.0 added support for reductions over explicit tasks. This expands the previous reduction support that was limited primarily to worksharing and parallel constructs. While the scope of a reduction operation in a worksharing construct is the scope of the construct itself, the scope of a task reduction can vary. This difference requires syntactical means to define the scope of reductions, e.g., the task_reduction clause, and to associate participating tasks, e.g., the in_reduction clause. Furthermore, the disassociation of the number of threads and the number of tasks creates space for different implementations in the OpenMP runtime. In this work, we provide insights into the behavior and performance of task reduction implementations in GCC/g++ and LLVM/Clang. Our results indicate that task reductions are well supported by both compilers, but their performance differs in some cases and is often determined by the efficiency of the underlying task management.

Keywords: OpenMP · reduction · worksharing · tasking.

1 Introduction

Since the first OpenMP specifications for Fortran and C/C++, the API has always included support for a defined set of reduction operations in worksharing and parallel constructs. This support covers C/C++ `for` and Fortran `do`

* This article has been authored by an employee of National Technology & Engineering Solutions of Sandia, LLC under Contract No. DE-NA0003525 with the U.S. Department of Energy (DOE). The employee owns all right, title and interest in and to the article and is solely responsible for its contents. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this article or allow others to do so, for United States Government purposes. The DOE will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (https://www.energy.gov/downloads/doe-public-access-plan). Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525.

loops with a defined iteration space, suitable for many iterative algorithms. OpenMP 3.0 introduced support for explicit task parallelism in OpenMP, enabling more irregular computations such as recursive algorithms and pointer chasing. That class of capabilities was further developed in subsequent versions of the specification to include the support of task dependencies in OpenMP 4.0 and the taskloop construct in OpenMP 4.5.

OpenMP 4.0 also added support for user-defined reductions over non-trivial data types and arbitrary operations. OpenMP 5.0 [8] finally brought support for explicit tasks to contribute to reductions, adding the means to define both the scope of such reductions and the tasks participating in them. As with many OpenMP features, implementors required time to incorporate support of task reductions, but both LLVM/Clang and GCC/g++ now include this feature.

This paper examines the current state of task reduction support in GCC/g++ and LLVM/Clang, including support of the language constructs as well as the performance of their implementations. For this purpose, we have selected three synthetic benchmarks that stress-test implementations and expose the cost of reduction support as well as tasking overheads for each compiler.

The rest of the paper is structured as follows. Section 2 provides background on reductions and currently supported syntax in OpenMP. Section 3 describes the benchmarks used in the evaluation, and Section 4 provides details of the experimental setup. Section 5 discusses performance results from the evaluation. Section 6 provides insight into reduction support in GCC/g++ and LLVM/-Clang and matches observed benchmark behavior with implementation choices. Section 7 surveys related work. Lastly, Section 8 summarizes this work and gives an outlook on further research directions.

2 Background

In this section we provide some context for our work. First, we consider general implementation strategies for reduction operations. Second, we give an overview of OpenMP task reductions.

2.1 Reductions and Their Common Implementation Strategies

In mathematical terms, a reduction algorithm is a numerical fold over a sequence of numbers. As the name suggests, it implies an iterative update (accumulation) of a result variable. For parallel formulations on parallel hardware in which the sequence of numbers is traversed concurrently and where the result variable is thus updated concurrently, data races can occur.

Two common strategies exist to avoid data races. One strategy makes memory updates atomic. The other creates thread-private data copies for the duration of the traversal of the sequence by each thread. In this strategy, a second step is required, during which the privatized copies are combined.

Atomic updates depend on software or hardware support of atomic memory updates and have implications for cache coherency traffic as a result of contention among threads to access the location of the reduction variable. Privatization avoids concurrent accesses to a memory location at the cost of private memory allocation, initialization, and the need for the final combination step. Which particular implementation is preferable on a given architecture depends on the size of the reduction variable and the access frequency.

OpenMP supports atomic memory accesses through the `atomic` construct which can be used directly by the developer to implement reductions. Programming model runtimes commonly rely on privatization for their internal implementation, which together with the possibility of using atomics, offers flexibility of choice to the programmer depending on the use case. Alternatively, the developer can implement privatization using the `threadprivate` directive. In this case, the developer is responsible for combining per-thread results as well.

From an OpenMP implementation perspective, privatization can be achieved by privatizing per thread or by privatizing per task. Privatization per task can incur significant overheads if the number of tasks is disproportionally larger than the number of threads. Since large numbers of tasks are common, implementations rely on per thread privatization where each task acquires the thread-private copy of the reduction variable at execution time.

Finally, the developer can avoid privatization or the need for atomic accesses in recursive task parallel programs by passing the reduction variable to each task as a function argument by value and returning the intermediate results as a return value. Unfortunately, the use of stack for the purpose of privatization is equivalent to per-task privatization and incurs the highest memory use and overheads due to repetitive initialization of stack variables. We call this approach *stack* in the evaluation section.

2.2 OpenMP Task Reduction Syntax and Semantics

The OpenMP specification admits the formulation of task reductions through *reduction scoping clauses* and a *reduction participating clauses.* The former "defines the region in which a reduction is computed", while the latter "specifies a task (or SIMD lane) as a participant in a reduction defined by a reduction scoping clause" [8].

The clauses are as follows, taking an operation *op* and reduction variable *var*.

– `reduction(task`, *op: var*`)`: Scopes a task reduction for a parallel or worksharing region
– `reduction(`*op: var*`)`: Scopes a task reduction for a taskloop region and makes the tasks created to execute the loop participants in the task reduction
– `task_reduction(`*op: var*`)`: Scopes a task reduction for a taskgroup region
– `in_reduction(`*op: var*`)`: Denotes participation of a task, target task, or taskloop in a task reduction

In Listing 1.1, a task reduction is scoped using the `task_reduction` clause on the `taskgroup` construct. Explicit tasks participating in the reduction use the `in_reduction` clause on the `task` construct. In Listing 1.2, the task reduction is scoped using the `reduction` clause of the `parallel` construct with the task modifier. As before, tasks bearing the in_reduction clause participate in the reduction. Finally, Listing 1.3 shows the use of the reduction clause on the taskloop construct, which acts as both a reduction scoping and reduction participating clause. References to the variable in the explicit tasks created by the OpenMP implementation to execute the iteration of the loop will all contribute to the reduction. Though not demonstrated here, the taskloop construct can also take an in_reduction clause to participate in a reduction already scoped in an enclosing region. The in_reduction clause can also be applied to a target construct, allowing a potentially offloaded target task to contribute to a task reduction.

```
#pragma omp parallel
#pragma omp single
#pragma omp taskgroup task_reduction(+: sum)
{
  #pragma omp task in_reduction(+: sum)
    sum += 1;
  #pragma omp task in_reduction(+: sum)
    sum += 2;
}
```

Listing 1.1: Simple Example of OpenMP Task Reduction

3 Benchmark Programs

To benchmark the implementations of OpenMP reductions, we consider a set of programs with distinct properties. They demonstrate the use of all OpenMP task reductions clauses and critically depend on an efficient implementation due to their high access frequency to the reduction variable.

In real-word applications, this frequency relative to other computation is significantly lower. This is the case where tasks are larger and spend more time in unrelated code. Instead, these benchmark applications stress-test implementations and show the hypothetical limitations, similar to roofline analysis.

For completeness, we contrast the OpenMP reduction support against other approaches to implement reductions such as atomics, user-managed thread-private copies, and returning partial values through the call tree. Prior to the availability of task reductions, these approaches would have been the only options for users attempting to combine results from OpenMP tasks.

3.1 Fibonacci

Fibonacci is a recursive program to calculate the *nth* number in the Fibonacci series. It represents a typical use case for task parallel programs where recursive formulations result in compact code or where an unknown iteration space at a given nesting level disallows the use of work sharing constructs.

```
int n, sum;

void fib (int n, int &sum)
{
  if (n < 2)
```

```
    sum += n;
  else
  {
    #pragma omp task in_reduction(+: sum)
      fib(n-1, sum);
    #pragma omp task in_reduction(+: sum)
      fib(n-2, sum);
  }
}

int main (int argc, char *argv[])
{
  n = atoi(argv[1]);

  #pragma omp parallel reduction (task, +: sum)
  #pragma omp single
  #pragma omp task in_reduction (+: sum)
    fib(n, sum);

  std::cout << "fib(" << n << ") = " << sum << std::endl;
  return 0;
}
```

Listing 1.2: Fibonacci calculation using OpenMP task reduction

Listing 1.2 shows an implementation using OpenMP task reductions. Here the program scopes the reduction using the `reduction` clause on the `parallel` construct with the `task` modifier, and tasks participating in that reduction scope use the `in_reduction` clause. Note that the reduction variable is passed by reference to the recursive function, because the reduction variable in the recursive function is not in the lexical scope of the parallel region. Further, using `taskwait` for synchronization is not required. The barrier at the end of the parallel region ensures that all tasks complete.

The program can be further augmented to provide "cut-off" values below which tasks would not be generated, thus coarsening parallelism. The effect is to reduce the number of tasks which in return lowers overheads of task creation and management. For example, if a cut-off of 10 were specified, then the calculation of $fib(9)$ would be handled by a direct sequential function call rather than an OpenMP task. Cut-offs could be either be implemented manually (if-then-else block) or using the `final` and `mergeable` clauses. Though not shown in the simplified code listing, for our evaluation we implemented manual cut-offs.

An alternative to task reductions would be the use of the `atomic` construct to update the reduction variable. In practice, this method is expected to introduce contention and limit effective parallelism as all threads compete to update the reduction variable.

Another alternative is to create thread-private copies of the reduction variable, accumulate partial sums in thread-local copies and combine the partial sums into the final sum at the end of the program. The `threadprivate` directive can be used to manage the copies. However, the addition of the final combining step makes this option somewhat cumbersome.

A third alternative with minimal requirements on compiler support is to transmit per-task partial sums as return values through the call stack. A drawback of this option is that a `taskwait` construct is needed at each nesting level in order to wait for the contributions of the child tasks.

3.2 Dot Product

Dot product implements the vector dot product of two arrays of numbers. Unlike Fibonacci, this benchmark is iterative rather than recursive. Listing 1.3 uses the `taskloop` construct to decompose the iteration space of the loop into tasks. In addition to task reduction, atomic, and thread-privatization versions, we also compare to a version using a worksharing construct with no explicit tasks.

For Fibonacci, the number of recursive function calls determines the number of tasks created and thus requires the use of cut-offs to limit the number of tasks. For Dot, the number of tasks is orthogonal to the algorithm itself and can be specified through the `num_tasks` clause on the `taskloop` construct.

```
#pragma omp parallel shared(x, y) num_threads(nthreads)
#pragma omp single
#pragma omp taskloop num_tasks(ntasks) reduction(+ : sum)
   for (unsigned long i = 0; i < n; ++i) {
      double tmp = x[i] * y[i];
      sum += tmp;
   }
```

Listing 1.3: Task reduction for vector dot product using the `taskloop` construct

3.3 Powerset

The Powerset benchmark computes the number of permutations of n elements by expanding a binary tree with a height of *log(n)*. While similar in algorithmic structure to Fibonacci, the Powerset produces a balanced tree which makes it less sensitive to task-stealing features in task schedulers. In addition to a variation of this algorithm using reduction variables of integer type, the Powerset benchmark also exercises implementations with user-defined reductions over a configurable type that is variable in size. We refer to it as *Powerset-UDR*. This configuration enables us to quantify and further differentiate the overheads originating from task management versus privatization. As with Fibonacci, we have implemented manual cut-offs (not shown in the simplified code listing).

```
int thr_priv_sum, cut_off;
#pragma omp threadprivate(thr_priv_sum)

void powerset(int n, int index) {
  for (int i = index; i < n; ++i){
#pragma omp task
    {
      powerset(n, i + 1);
      thr_priv_sum++;
    }
  }
}

int main(int argc, char *argv[]) {
  int n = atoi(argv[1]);
  int nthreads = atoi(argv[2]);
  cut_off = atoi(argv[3]);
  int sum = 0;

#pragma omp parallel num_threads(nthreads)
  {
    thr_priv_sum = 0;
#pragma omp single
#pragma omp task
    powerset(n, 0);
  }

//Reduce thread-private copies
#pragma omp parallel num_threads(nthreads)
  {
#pragma omp single
    nthreads = omp_get_num_threads();
#pragma omp for reduction(+ : sum)
    for (int i = 0; i < nthreads; i++)
      sum += thr_priv_sum;
  }

  std::cout << "powerset(" << n << ") = " << sum << std::endl;
  return 0;
}
```

Listing 1.4: Powerset using thread-private reduction variables obtained with manual thread privatization

The Powerset benchmark includes the variations described in the previous section for Fibonacci. Listing 1.4 shows the implementation of Powerset using the `threadprivate` directive and the subsequent manual reduction of private copies into the final reduction variable.

4 Experimental Setup

The test machine comprises Intel® Xeon® “Skylake" Platinum 8160 Processors in a dual socket configuration with 24 cores per socket (48 cores total) and 2 hardware threads per core running at 2.1 GHz. The memory is 192 GB DDR4. The operating system is Red Hat® Enterprise Linux® 7.9. The compiler and runtime versions are LLVM/Clang 14.0 (release) and GCC 13 (not yet released, code version dated 20220518).

For both compilers we have used the following sequence of options *-fopenmp, -Wall, -Wextra, -pedantic, -Werror* and *-O3*. Further, we have set the environment variables *OMP_PROC_BIND* and *OMP_PLACES* to *close* and *cores* respectively during execution. On the test machines, this results in a thread mapping of one thread per core.

5 Evaluation

We have evaluated the set of presented benchmarks for the various implementations, using a variable number of threads ranging from one to 128 and for a variable number of tasks. Further, we have compiled all implementations of all benchmarks with both the LLVM/Clang and the GCC/g++ compilers using the same compiler options. Lastly, the evaluation of a version of Powerset using user-defined reductions includes results for variable reduction type sizes. This section summarizes key finding and provides representative figures for configurations with 48 threads only. Executions with smaller thread counts exhibit similar behavior, while executions with more than 48 threads result in non-representative data due to over-subscription of the system and the resulting effects. All benchmark results show the average total execution time of 5 repetitions for a respective constant problem size and given range of tasks.

Figures 1 and 2 show performance results for the Fibonacci and Powerset computations. Key insights for these two benchmarks are as follows.

- Performance of implementations using the OpenMP language features for reductions is the same order of magnitude as prior available implementations using stack-local variables (*stack*) or manual privatization (*threadprivate*).
- The use of `parallel` with the `task` modifier (*parallel-task-red*) yields similar results to the use of the `task_reduction` clause on the `taskgroup` construct (*taskgroup-red*).
- Atomic accesses (*atomic*) incur high overhead regardless of the number of tasks due to threads contending for the same memory location.
- Implementations relying on stack-local variables and the *taskwait* construct depend on the efficiency of the underlying tasking implementation. GCC/g++ performs well only for low task counts, while LLVM/Clang outperforms for large task counts.
- Implementations using the *threadprivate* clause to manually privatize variables (*threadpriv*) underperform compared to other techniques for small task counts. Recall that they incur the cost of the additional step of manually combining the thread-private copies.
- When using GCC/g++, the performance of `taskloop` reductions (*taskloop-red*) degrades with large numbers of tasks.
- No significant differences were observed when using the `untied` task modifier (*parallel-task-red-untied*) compared to the tied default (*parallel-task-red*).

Figure 3 shows results for the dot-product with two to 131k tasks for LLVM/-Clang and an input problem size of 2^{24} values per array. This input size corresponds to array allocations of 128MB each. The results indicate a similar behavior for the atomic implementation as described for Fibonacci and Powerset. The implementation using the `parallel for` construct uses no tasks and is provided for reference. As it is invariant to the number of tasks, its performance represents a horizontal line (*parallel-for-red*). Lastly, all other techniques perform similarly: Performance degrades when the number of tasks is too low to provide enough parallelism and when the number of tasks is too large with the resulting granularity being too fine. However, the amount of work per task is significantly higher compared to Powerset or Fibonacci, potentially underexposing some technique-specific performance variations. We have observed comparable performance for GCC/g++ on this benchmark.

The graph in Figure 4 shows results for the Powerset benchmark using user-defined reductions with a constant number of 262k tasks for LLVM/Clang and GCC/g++. Results indicate that all techniques except *stack* are invariant to the size of the reduction variable, and thus the cost of memory allocation is equal. For LLVM/Clang, the implementation using *stack* degrades in performance with increasing type sizes due to increasingly distant memory accesses for stack operations. Results for GNU/g++ resemble performance results shown in Figure 2b corresponding to 262k tasks: In both those results and the results for UDRs of all sizes, tasking overheads dominate.

To summarize, the performance of task reductions is determined by the task granularity and task count, by properties of a reduction technique and by its implementation in the runtime system.

Techniques available prior to the support of task reductions in OpenMP vary significantly in performance. For higher degrees of concurrency and frequent accesses to the reduction variable, atomics achieve the lowest performance. The use of stack-local variables requires task synchronization and relies on efficient task management in the runtime. Finally, manual privatization using `threadprivate` variables requires a final reduction of all private copies once the reduction completes. If the final reduction incurs additional overhead, performance degrades.

Task reduction support in OpenMP using the `parallel` and `taskgroup` constructs exhibits performance asymptotic to the *threadprivate* version, suggesting that both LLVM/Clang and GCC/g++ internally use per-thread privatization with tasks acquiring and reusing such thread-private allocations. In this case, internal optimizations can raise the performance beyond that of manual privatization coded at user level. In particular, the implementation does not need to expose OpenMP semantics for its internal mechanism used to combine results and may employ a more sophisticated reduction of thread-private copies such as an in-line parallel tree based reduction.

The next section examines implementations of task reduction in both LLVM/-Clang and GCC/g++, as well as relevant differences in their general task management approaches.

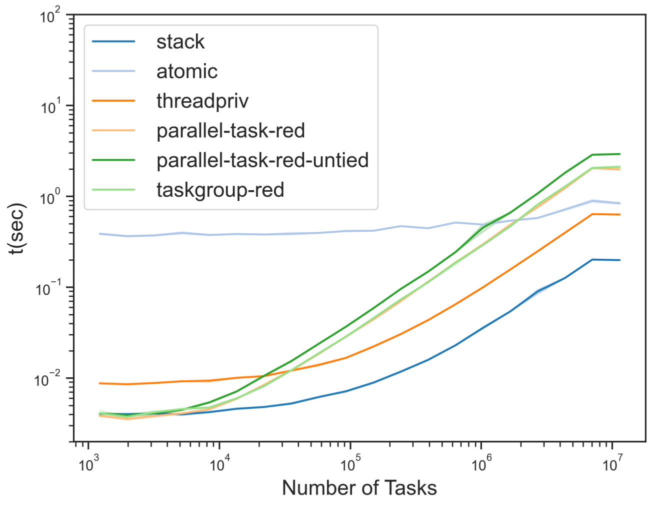
6 Implementations in GCC and LLVM/Clang

The understanding of performance characteristics described in the previous section requires inspection of tasking and task reduction support in the front-end compiler as well as the runtime. Of particular interest is the implementation of memory privatization and whether it occurs on thread or task level.

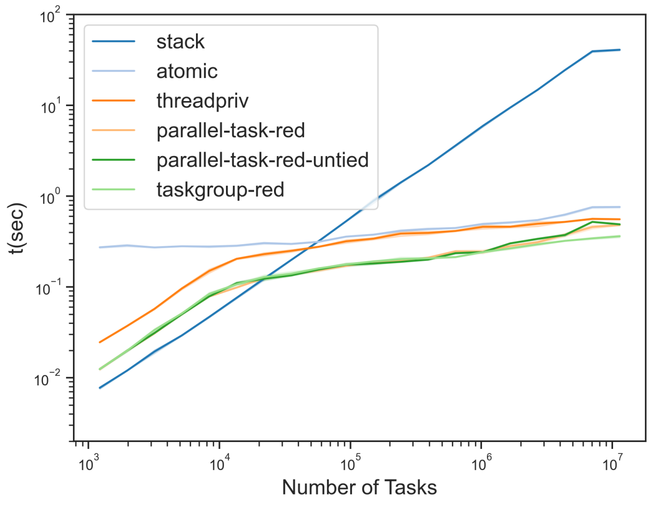
Figure 1.5 shows example code for a task participating in a task reduction. Figure 1.6 shows the corresponding intermediate code representation produced by the GCC/g++ (compiler *-fdump-tree-optimized*). Accesses to the original memory location are redirected to a new memory location obtained by calling *__builtin_GOMP_tas_reduction_remap*. This function obtains the associated thread-private memory location corresponding to the reduction variale registered by the reduction clause.

```
void func(int &sum) {
#pragma omp task in_reduction(+ : sum)
    sum++;
}
```

Listing 1.5: Sample code for a task participating in a reduction

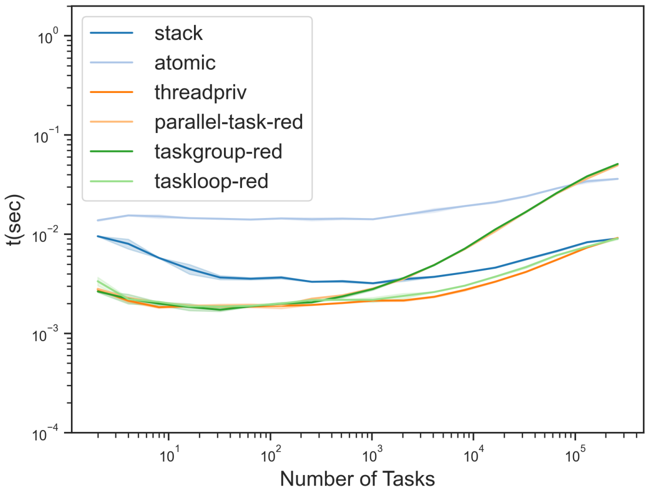


(a) LLVM/Clang

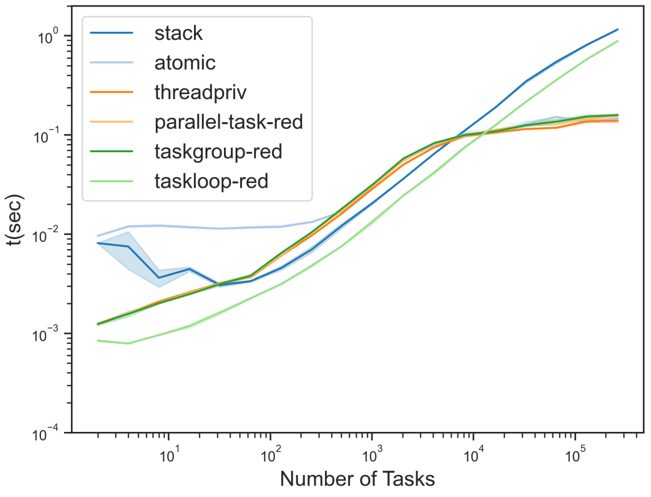


(b) GCC/g++

Fig. 1: Fibonacci computation with a constant problem size of *N=33*, 48 threads and a variable task cutoff resulting in a range of 1.2k-11405k tasks, showing differences between compilers and techniques



(a) LLVM/Clang



(b) GCC/g++

Fig. 2: Powerset computation with a constant problem size of *N=18*, 48 threads and a variable cutoff with a range of 2-262k tasks, compiled with both compilers

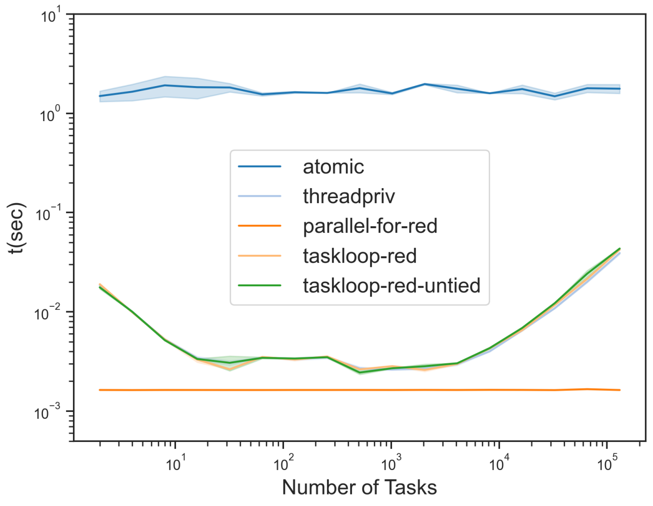


Fig. 3: Dot-product compiled with LLVM/Clang with a constant problem size of N=2^{24}, 48 threads and a variable cutoff resulting in a range of 2-131k tasks

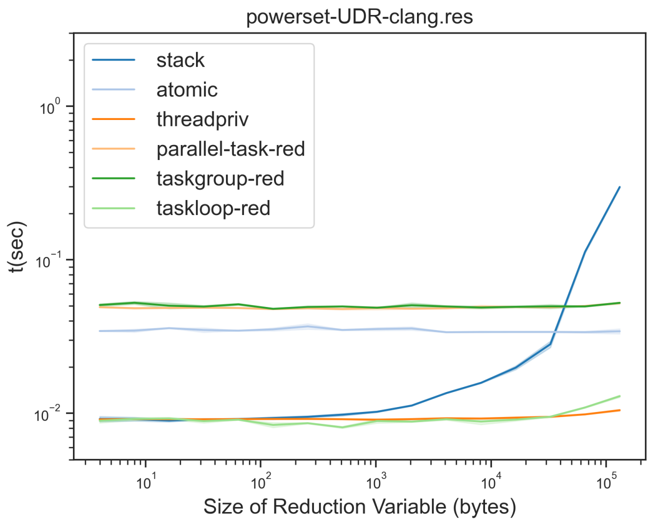
```
void func (int & sum) {
  struct .omp_data_s.0 .omp_data_o.1;
  ...
  .omp_data_o.1.sum = sum_2(D);
  __builtin_GOMP_task (_Z4funcRi._omp_fn.0, &.omp_data_o.1, 0B, 8, 8, 1,
     0, 0B, 0, 0B);
  return;
}

void _Z4funcRi._omp_fn.0 (struct .omp_data_s.0 & restrict .omp_data_i) {
  ...
  void * D.2516[1];
  _3 = .omp_data_i_2(D)->sum;
  D.2516[0] = _3;
  __builtin_GOMP_task_reduction_remap (1, 0, &D.2516);
  sum_6 = D.2516[0];
  _10 = *sum_6;
  _11 = _10 + 1;
  *sum_6 = _11;
  return;
}
```

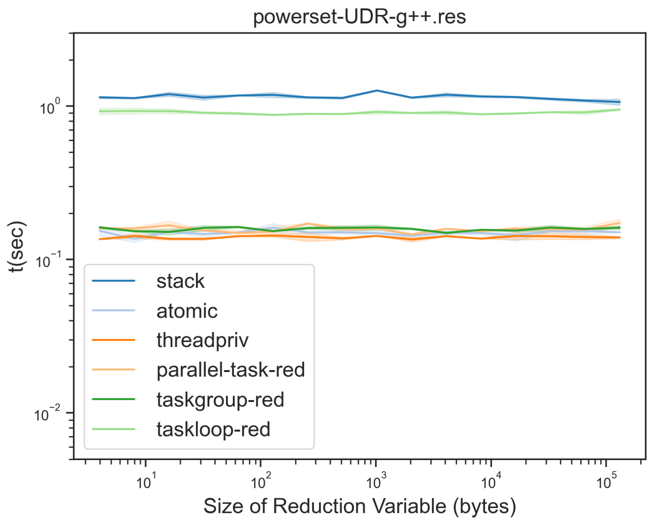
Listing 1.6: Intermediate code fragments generated by the GCC/g++ front-end compiler for the example code in Figure 1.5

LLVM/Clang supports task reductions through per-thread privatization as well. Similar to the approach in GCC, the intermediate code calls the function *__kmpc_task_reduction_get_th_data* to access the thread-private copy.

The overhead costs of task management are critical to performance both with and without task reductions. The use of per-thread task queues in the LLVM runtime contributes to lower task management costs compared to the



(a) LLVM/Clang



(b) GCC/g++

Fig. 4: Powerset with a constant problem size of $N{=}18$, 48 threads, 262k tasks and variable reduction type with type size range of 4B-131KB

GCC runtime with its centralized queue that is shared among all threads in the team. High overhead costs particularly impact our *stack* benchmark versions that pass partial results through the call stack, because they require taskwait synchronizations that induce additional accesses to task queues in the runtime.

7 Related Work

Prior to the addition of task reductions in OpenMP 5.0, evaluation studies [4, 2] of the proposed feature had been demonstrated using the Nanos runtime system[1] and Mercurium compiler[2] [1]. In addition to OpenMP tasking, they implement the OmpSs programming model[3] [5], a tasking-centric programming model with close ties to OpenMP and support for task reductions. Previous work also explored array reductions over OmpSs tasks [3]. User-defined reductions for OpenMP were proposed by Duran et. al [6]. Reductions in other task parallel languages and language extensions include X10/Habanero-Java phaser accumulators [11] and finish accumulators [10], as well as Cilk++ hyperobjects [7]. Blaze-Tasks is a C++17-based framework for task scheduling and reductions [9].

8 Conclusions and Future Work

Our study provides evidence that the task reduction features in OpenMP are well supported by GCC and LLVM/Clang today. Performance insights indicate that the use of the language features is meaningful and provides performance comensurate to efficient manually implemented reductions for reasonable task sizes. For reproducibility, we intend to make available the benchmarks, the scripts to build and run them, and the complete set of graphs, upon approval.

Key topics that warrant further investigation are performance differences among compilers and the efficiency of their support for `taskwait` synchronizations (stressed by our manually-coded stack-based reductions) and reductions on `taskloop` constructs. Further inspection of their implementations along with a deeper experimental evaluation is a subject for future work. Based on the results in this paper, our recommendation to users is that they can confidently employ the convenience and performance of task reductions for their OpenMP applications on multicore CPUs.

References

1. Balart, J., Duran, A., Gonzàlez, M., Martorell, X., Ayguadé, E., Labarta, J.: Nanos Mercurium: a research compiler for OpenMP. In: European Workshop on OpenMP (EWOMP04). pp. 103–109 (2004)

[1] https://pm.bsc.es/nanox
[2] https://pm.bsc.es/mcxx
[3] https://pm.bsc.es/ompss

2. Ciesko, J., Mateo, S., Teruel, X., Beltran, V., Martorell, X., Badia, R.M., Ayguadé, E., Labarta, J.: Task-parallel reductions in OpenMP and OmpSs. In: DeRose, L., de Supinski, B.R., Olivier, S.L., Chapman, B.M., Müller, M.S. (eds.) 10th International Workshop on OpenMP (IWOMP 2014). Lecture Notes in Computer Science, vol. 8766, pp. 1–15. Springer (Sep 2014)
3. Ciesko, J., Mateo, S., Teruel, X., Martorell, X., Ayguadé, E., Labarta, J.: Supporting adaptive privatization techniques for irregular array reductions in task-parallel programming models. In: Maruyama, N., de Supinski, B.R., Wahib, M. (eds.) 12th International Workshop on OpenMP (IWOMP 2016). Lecture Notes in Computer Science, vol. 9903, pp. 336–349. Springer International Publishing, Cham (2016)
4. Ciesko, J., Mateo, S., Teruel, X., Martorell, X., Ayguadé, E., Labarta, J., Duran, A., de Supinski, B.R., Olivier, S., Li, K., Eichenberger, A.E.: Towards task-parallel reductions in OpenMP. In: Terboven, C., de Supinski, B.R., Reble, P., Chapman, B.M., Müller, M.S. (eds.) 11th International Workshop on OpenMP (IWOMP 2015). Lecture Notes in Computer Science, vol. 9342, pp. 189–201. Springer International Publishing (2015)
5. Duran, A., Ayguadé, E., Badia, R., Labarta, J., Martinell, L., Martorell, X., Planas, J.: OmpSs: a proposal for programming heterogeneous multi-core architectures. Parallel Processing Letters **21**(02), 173–193 (2011)
6. Duran, A., Ferrer, R., Klemm, M., de Supinski, B.R., Ayguadé, E.: A proposal for user-defined reductions in OpenMP. In: Sato, M., Hanawa, T., Müller, M.S., Chapman, B.M., de Supinski, B.R. (eds.) 6th International Workshop on OpenMP (IWOMP 2010). Lecture Notes in Computer Science, vol. 6132, pp. 43–55. Springer, Berlin, Heidelberg (2010)
7. Frigo, M., Halpern, P., Leiserson, C.E., Lewin-Berlin, S.: Reducers and other Cilk++ hyperobjects. In: Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures (SPAA'09). pp. 79–90. ACM, NY, NY, USA (2009)
8. OpenMP Architecture Review Board: OpenMP Application Programming Interface Version 5.0. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf (Nov 2018)
9. Pirkelbauer, P., Wilson, A., Peterson, C., Dechev, D.: Blaze-Tasks: A framework for computing parallel reductions over tasks. ACM Trans. Archit. Code Optim. **15**(4) (Jan 2019)
10. Shirako, J., Cavé, V., Zhao, J., Sarkar, V.: Finish accumulators: An efficient reduction construct for dynamic task parallelism. In: Kasahara, H., Kimura, K. (eds.) 25th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2012). Lecture Notes in Computer Science, vol. 7760, pp. 264–265. Springer (2012)
11. Shirako, J., Peixotto, D.M., Sarkar, V., Scherer, W.N.: Phaser accumulators: A new reduction construct for dynamic parallelism. In: IEEE International Symposium on Parallel and Distributed Processing (IPDPS 2009). pp. 1–12. IEEE, Rome, Italy