SAND2022-13337C

Sandia National Laboratories

Exceptional service in the national interest

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

Jan Ciesko, Stephen Olivier

Thursday, 29th September, 2022

UT, Chattanooga, Tennessee,

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 ...ration under contract DE-NA0003525.

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & 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-NA0003525.

U.S. DEPARTMENT OF ENERGY

NNSA

Agenda

- Reductions in OpenMP
- Implementations in LLVM/Clang and GCC
- Benchmarks
- Evaluation
- Conclusion

Reductions in OpenMP

- OpenMP supports reductions since V1.0
- Tasking since V3.0 in 2008
- Demand for task-parallel reductions
- I started working on this in 2013
- Presented to OMP LC in 2015.
- Proposal made it into the spec in 2018 (OpenMP 5.0)
- Task reductions are conceptually concurrent and are orthogonal to the depend clause
- Compiler support evaluation is important for performance portability
  - How well did we do in the spec?
  - Did implementers implement the spec?

OpenMP C and C++ Application Program Interface

Version 1.0 – October 1998

2.3 `parallel` Construct

The following directive defines a *parallel region*, which is a region of the program that is to be executed by multiple threads in parallel. This is the fundamental construct that starts parallel execution.

```
#pragma omp parallel [clause[ clause] ...] new-line
        structured-block
```

The *clause* is one of the following:

`if(`*scalar-expression*`)`

`private(`*list*`)`

`firstprivate(`*list*`)`

`default(shared | none)`

`shared(`*list*`)`

`copyin(`*list*`)`

`reduction(`*operator*`:` *list*`)`

Reductions in OpenMP

- Task reductions allow reductions computed by arbitrary task graphs
- Challenge: definition of participating tasks and of the scope of the reduction computations
- Clauses for task reductions:

Clause	Semantic	Description
reduction(task, op: var)	Scoping, Participation	Scopes a task reduction for a **parallel or work-sharing region***.
reduction(op: var)	Scoping, Participation	Scopes a task reduction for a **taskloop** region and makes created tasks participants.
task_reduction(op: var)	Scoping	Scopes a task reduction for a **taskgroup** region.
in_reduction(op: var)	Participation	Denotes participation of a **task**, **target task**, or **taskloop** in a task reduction.

- ‘task’ reduction modifier on construct only for ‘parallel’, ‘for’, ‘sections’ or ‘scope

Rule of Thumb:

1. Participating tasks must have a enclosing scope that defines a task reduction
2. The reduction computation completes by the time the scope ends

Implementations in GCC and LLVM/Clang

- Privatization and data reuse

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

gcc -fopenmp -c task.c -fdump-tree-optimized -o task.o.gcc

```
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];    // new double pointer
  _3 = .omp_data_i_2(D)->sum; // reference to original reduction storage location
  D.2516[0] = _3;
  __builtin_GOMP_task_reduction_remap (1, 0, &D.2516);   // redirect
  sum_6 = D.2516[0]; // dereference
  _10 = *sum_6;
  _11 = _10 + 1;   // use
  *sum_6 = _11;
  return;
}
```

clang -Xclang -S -emit-llvm -Xpreprocessor -fopenmp -c task.c -o task.o.s.llvm (not shown)

*GCC Version 13, 20220518

Benchmarks

Applications*

- Fibonacci
- Powerset
- Powerset-UDR
- Dot-product

6 implementations each:

- Parallel task-reduction
- Taskloop reduction
- Taskgroup reduction
- Manual per-thread data privatization
- Atomics
- Stack

README.md

OMP Task Bench (OMP-TB)

OMP-TB is a collection of benchmarks to measure tasking performance and tasking-related features in OpenMP. Currently it includes benchmark as listed below. Benchmarks in the `reductions` sub-directory target task-parallel reduction support. In general, such benchmarks are useful to evaluate compiler language support as well as its efficient implementation.

OMP-TB Benchmarks

- reductions/dot (Dot Product)
- reductions/fib (Fibonacci)
- reductions/powerset (Powerset Permutations)
- reductions/powerset-final (Powerset Permutations using the final OpenMP clause)
- reductions/powerset-UDR (Powerset Permutations using user-defined reductions)
- reductions/others/array_sum (Array Sum)
- reductions/others/knapsack (Knapsack)
- reductions/others/knightstour (Knights Tour)
- reductions/others/max_height_tree (Max Height)
- reductions/others/nbinarywords (n-Permutations)
- reductions/others/nqueens (N-Queens)
- reductions/others/TSP (Travelling Salesman Problem)

[1] https://github.com/sandialabs/openmp_task_bench

* OMP-TB [1] contains further examples

Implementation: Fibonacci

- Parallel task reduction using manual cut-off

In results: “parallel-red”

main

```
...
#pragma omp parallel reduction(task, + : sum) \
                       num_threads(conf.num_threads)
#pragma omp single
#pragma omp task firstprivate(n) in_reduction(+ : sum)
    fib(n, sum);
```

task

```
void fib(int n, int &sum) {
  if (n < 2)
    sum += n;
  else {
    if (n < cut_off) {
      fib(n - 1, sum);
      fib(n - 2, sum);
    } else {
#pragma omp task firstprivate(n) in_reduction(+ : sum)
      fib(n - 1, sum);

#pragma omp task firstprivate(n) in_reduction(+ : sum)
      fib(n - 2, sum);
    }
  }
}
```

Note: we avoided the use of final

Implementation: Fibonacci

- Taskgroup reductions using manual cut-off

In results: “taskgroup-red”

```
...
#pragma omp parallel shared(n, sum) num_threads(conf.num_threads)
#pragma omp single
#pragma omp taskgroup task_reduction(+ : sum)
#pragma omp task firstprivate(n) in_reduction(+ : sum)
  fib(n, sum);
```

main

```
void fib(int n, int &sum) {
  if (n < 2)
    sum += n;
  else {
    if (n < cut_off) {
      fib(n - 1, sum);
      fib(n - 2, sum);
    } else {
#pragma omp task firstprivate(n) in_reduction(+ : sum)
      fib(n - 1, sum);

#pragma omp task firstprivate(n) in_reduction(+ : sum)
      fib(n - 2, sum);
    }
  }
}
```

task

Implementation: Fibonacci

- Using the stack and manual cut-off

In results: “stack”

```
int fib(int n) {
  int x, y;

  if (n < 2)
    return n;
  else {
    if (n < cut_off) {
      x = fib(n - 1);
      y = fib(n - 2);
    } else {
#pragma omp task shared(x) firstprivate(n)
      x = fib(n - 1);

#pragma omp task shared(y) firstprivate(n)
      y = fib(n - 2);

#pragma omp taskwait
    }
    return x + y;
  }
}
```

task

```
...
#pragma omp parallel shared(sum) num_threads(conf.num_threads)
#pragma omp single
#pragma omp task shared(sum) firstprivate(n)
    sum = fib(n);
```

main

Implementation: Fibonacci

- Using explicit threadprivate and manual cut-off

task

```c
#pragma omp threadprivate(mysum)
void fib(int n) {
  if (n < 2)
    mysum += n;
  else {
    if (n < cut_off) {
      fib(n - 1);
      fib(n - 2);
    } else {
#pragma omp task firstprivate(n)
      fib(n - 1);

#pragma omp task firstprivate(n)
      fib(n - 2);
    }
  }
}
```

In results: “threadpriv”

main

```c
...
#pragma omp parallel num_threads(conf.num_threads)
{
      mysum = 0;
#pragma omp single
#pragma omp task
      fib(n);
}

#pragma omp parallel num_threads(conf.num_threads)
{
#pragma omp single
      nthreads = omp_get_num_threads();
#pragma omp for reduction(+ : sum)
      for (int i = 0; i < nthreads; i++)
        sum += mysum;
}
```

Evaluation

System configuration:

- **LLVM/Clang 14.0 (release)**
- **GCC 13 (code version dated 20220518)**
- Intel® Xeon® Skylake Platinum 8160 Processor, dual-socket, 48 cores, (blake.sandia.gov)
- 192GB RAM
- Flags: -fopenmp, -Wall, -Wextra, -pedantic, -Werror, -O3.
- Env: OMP_PROC_BIND = close, OMP_PLACES = cores (one thread per core with incremental core IDs)

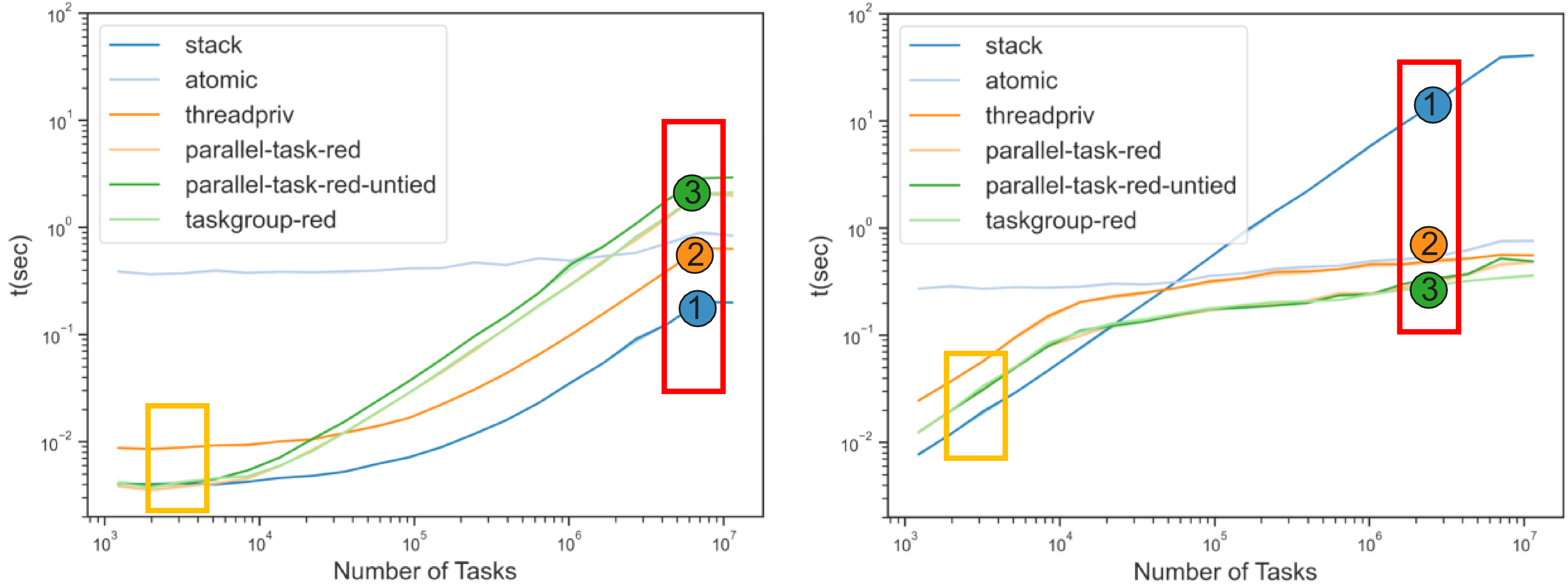
Methodology

- Each benchmark ran 5x, average time and standard deviation recorded
- Generate very large task counts to accumulate overheads
- Downside: tasking vs reduction overheads blend together

Evaluation: Fibonacci

1) Stack has lowest OH, OH with # tasks, has TW
2) TP has low OH but requires manual reduction
3) Comparable performance

- Time over number of tasks, N=33, 48 threads, variable cut-off, **1.2k – 11405k tasks**

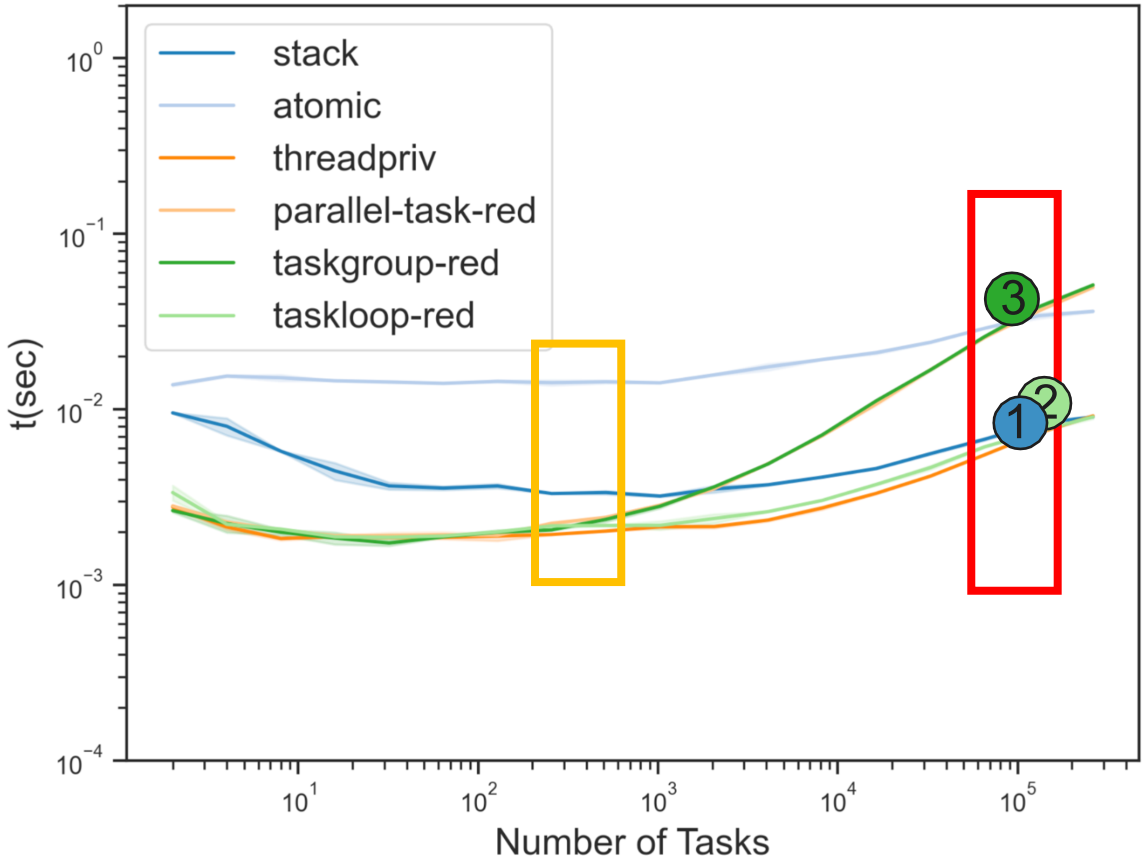


(a) LLVM/Clang (b) GCC/g++

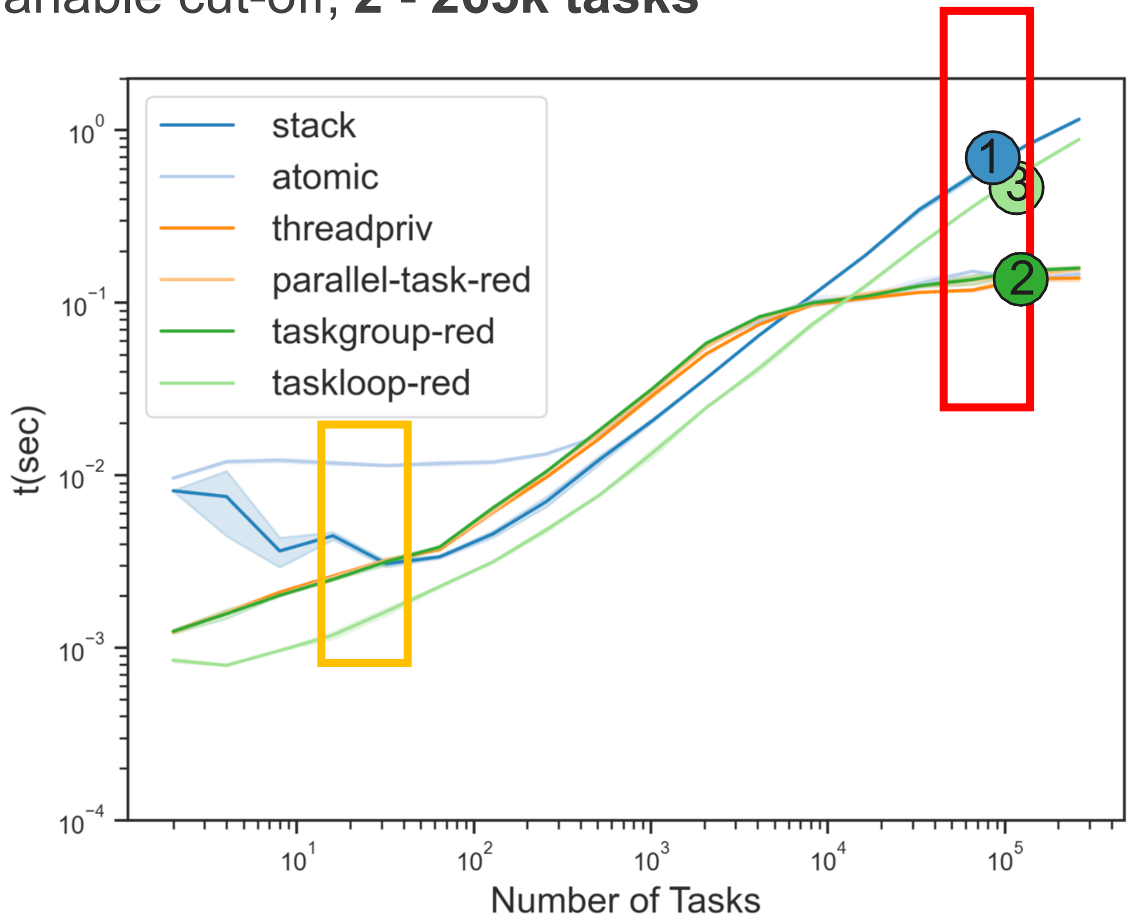
Evaluation: Powerset

1) Similar behavior of taskwait to Fib
2) Taskgroup and taskloop diverge (3), due to differences in reduction list item lookup and explicit task creation (taskloop)

- Time over number of tasks, N=18, 48 threads, variable cut-off, **2 - 265k tasks**



(a) LLVM/Clang

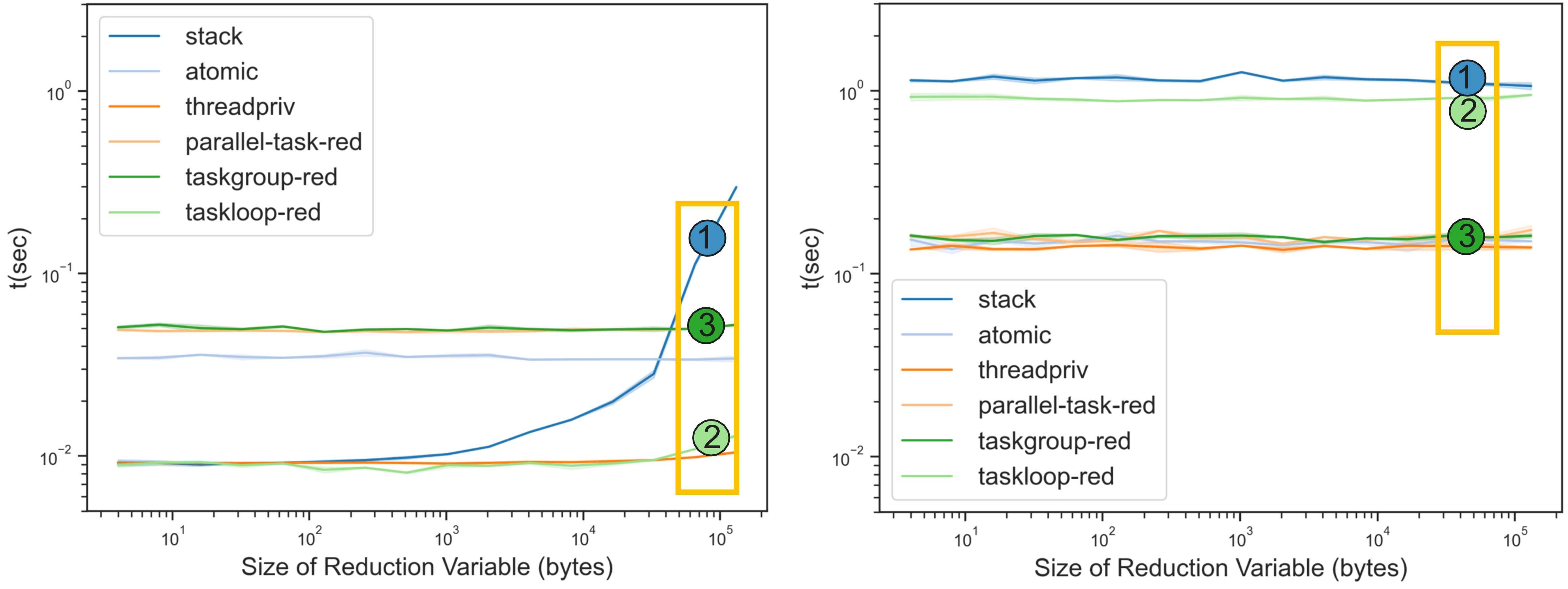


(b) GCC/g++

Evaluation: Powerset **UDR**

1) Stack accesses causing cache faults (LLVM)
2) Differences in reduction list item look-up and explicit task creation (taskloop)

- Time over reduction size (UDR), **4B – 131kB**, 48 threads, 262k tasks

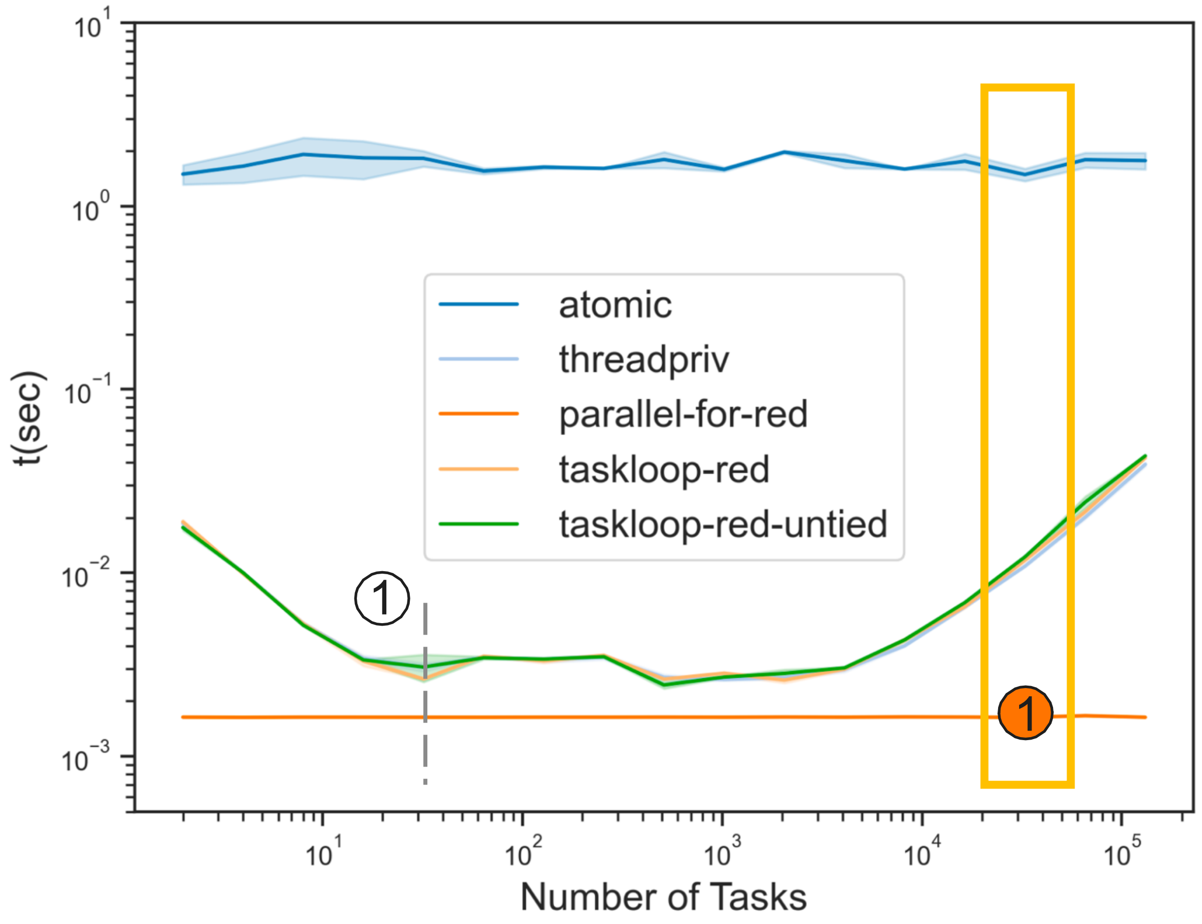


(a) LLVM/Clang (b) GCC/g++

Evaluation: Dot-product (not recursive)

1) Execution with work for 48 threads
2) Parallel for is invariant as it does not use tasks

- Time over number of tasks, N=2^{24}, 128MB, 48 threads, variable cut-off, **2 - 131k tasks**



Conclusion

- Both compilers support task-parallel reductions as in the spec
- Performance is comparable to manually implemented and optimized reductions
- For large task counts, tasking overheads dominate (incl. taskwait)
- We recommend the use of these constructs

Future work

- Evaluation of tasking implementations in LLVM/Clang and GCC

Links:

[1] https://github.com/sandialabs/openmp_task_bench