Support for OpenMP parallel loops in Tapenade

Laurent Hascoët
laurent.hascoet@inria.fr

Jan Hückelheim
jhueckelheim@anl.gov
Motivation

• OpenMP is widely used to parallelize C, C++, Fortran
• Several papers and PhD theses presented theory, implementations for specific use cases, prototype tools...
• Obviously, there is demand. Users want this to “just work”
• In forward and reverse mode
OpenMP crash course

• Consider this sequential loop to compute the dot product of a and b

```fortran
function dotprod(a,b)
    integer :: i, n
    real, dimension(n) :: a, b
    real :: dotprod
    dotprod = 0.0
    do i=1,n
        dotprod = dotprod + a(i) * b(i)
    end do
end function
```
OpenMP crash course

• The same thing, in parallel

```fortran
function dotprod(a,b)
  integer :: i, n
  real, dimension(n) :: a, b
  real :: dotprod dotprod = 0.0

  !$omp parallel do reduction(+:dotprod)
  do i=1,n
    dotprod = dotprod + a(i) * b(i)
  end do
end function
```
OpenMP crash course

• A bit more verbose, with scopes and schedule spelled out explicitly

```fortran
function dotprod(a,b)
integer :: i, n
real, dimension(n) :: a, b
real :: dotprod dotprod = 0.0
!$omp parallel do reduction(+:dotprod) &
!$omp& schedule(static) private(i) shared(n,a,b)
do i=1,n
    dotprod = dotprod + a(i) * b(i)
end do
end function
```
Contributions and Limitations

• Latest Tapenade release can handle OpenMP parallel worksharing loop construct: !$omp parallel do

• Supports most scoping clauses: shared, private, firstprivate, lastprivate, reduction(+)

• Supports all schedules: static, dynamic, guided, auto, default

• So far: no other pragmas allowed. No tasks, target offloading, critical sections, or anything else. Just parallel worksharing loops.

• Most other features appear doable and interesting, “just” needs more time and work.

• Details in ACM TOMS article, yet to appear. Ask for a copy if interested.
Main challenges

• Everything is easy in forward mode:
  • Just AD as usual
  • Tangent variables are scoped like primal variables

• Main challenges in reverse mode:
  1. Scheduling
  2. Scoping
1. Scheduling

• Which thread will work on which iteration(s)?
• This affects data flow.
• Iterations can depend on each other, even in parallel loops.
1. Scheduling example

• Schedule can affect result in arbitrary ways

```python
k = 0
%!omp parallel do firstprivate(k)
do i=1, 10
  if(k .eq. 0) then
    ! do something in each thread’s first iteration
    res(i) = omp_get_thread_num()
  else
    ! do something else in subsequent iterations
    res(i) = -1
  end if
k = 1
end do
```
1. Scheduling summary

- Ensure corresponding primal and adjoint iterations execute on same thread, in reverse order
- Override OpenMP static scheduler to control static schedule
- Record, revert, replay dynamic schedules
- Push/pop stack must be threadprivate
2. Scoping

• For each primal variable, need to determine correct scope of adjoint
• Plenty of previous work covers selected aspects, e.g. Michael Förster’s PhD thesis
• As far as we know, previous work has not covered all possible primal scopes and expressed their adjoints in terms of OpenMP scopes
• Scoping affects data flow, important to get it right
2. Scoping example

• What does this print?

```plaintext
k = 0
!$omp parallel do private(k)
do  i=1, 10
   k = 1
end do
print *, k
```
2. Scoping example

• What does this print?

```plaintext
k = 0
!
omp parallel do private(k)
do  i=1, 10
   k = 1
end do
print *, k
```

• If compiled without OpenMP support, prints “1”: k inside the loop is the same memory as k outside the loop

• If compiled with OpenMP support, prints “0”: Each thread has its own k, which is distinct from k outside the loop.
Shared variables require special handling, depending on circumstances.
Performance example

Fig. 19. **Left:** Absolute runtimes for GFMC test case. The atomic parallel time is 376 s and is truncated in the plot. **Right:** Scalability for primal and tangent GFMC is equally good but is significantly worse for adjoint reductions. The adjoint program using atomics scales well but performs too poorly compared with the serial adjoint program to show up in the plot.

Atomics or reductions can severely affect performance.
SPRAY: Sparse Reductions for Arrays in OpenMP

Jan Hückelheim
jhueckelheim@anl.gov

Johannes Doerfert
jdoerfert@anl.gov
Alternatives to atomics or reductions?

• Parallel adjoints, and many other programs, may contain accesses like:

```c
!$omp parallel do
do  i=1, n
    res(idx(i)) += ...
end do
```

• If `idx(i)` can be the same for two threads, we have a data race.
• We can solve this with atomic updates (slow) or reductions (slow and memory hungry)
• We wrote SPRAY to solve this
Separation of concerns

• The programmer (or AD tool) should only worry about high level semantics: “accumulate these contributions to shared array”

• SPRAY backends receive contributions, and add them in efficient and thread-safe manner

• Different backends for different access patterns, hardware platforms, problem sizes, etc

• Strategies include atomics and reductions, block-wise privatization, queues, and others
Small usage example

• Array is wrapped in SPRAY reducer object:

```cpp
res_s = spray::KeeperReduction(res)
!$omp parallel do reduction(+:res_s)
do i=1, n
    res_s(idx(i)) += ...
end do
```

• Inside parallel region, reducer object is used instead of original array
• SPRAY ensures that updates are correctly applied after parallel region
• Easily switch strategies, e.g. BlockReduction instead of KeeperReduction
• See our IPDPS 2021 paper: https://ieeexplore.ieee.org/document/9460492
Small performance example

Fig. 11. Speedup of built-in OpenMP and selected SPRAY reduction implementations over the sequential convolution back-propagation, on Intel, GNU, and LLVM compilers. All parallel implementations have some overhead. Hence, their lines start below 1, indicating a slowdown, and rise above 1, indicating a speedup as more threads are added. The sequential version is created with each compiler; hence the baseline performance differs between subplots, and better scalability with one compiler does not imply better absolute run time.
Questions?

• Jan Hückelheim: jhueckelheim@anl.gov
• Laurent Hascoët: laurent.hascoet@inria.fr
• Johannes Doerfert: jdoerfert@anl.gov

Acknowledgements

The Tapenade OpenMP work was funded in part by support from the U.S. Department of Energy, Office of Science, under contract DE-AC02-06CH11357. We gratefully acknowledge the computing resources provided and operated by the Joint Laboratory for System Evaluation (JLSE), at Argonne National Laboratory. The SPRAY work was additionally in part supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering, and early testbed platforms, in support of the nation’s exascale computing imperative.