



OpenMP Doacross Loops in Practice: Mini-Applications Case Study

September 18, 2017
Gabriele Jost and Henry Jin

Outline



- Background
 - The OpenMP doacross concept
- LU-OMP implementations
 - Different algorithms
 - Manual synchronization vs doacross
- Performance Analysis:
 - Synchronization
 - Work scheduling
 - Compilers
- Summary and Conclusions
- Time permitting:
 - Synchronization Pure: The Sync_p2p Parallel Research Kernels



Background

- OpenMP 4.0:
 - **ordered** clause for the work sharing construct
 - **ordered** construct to enclose a structured block within the loop body
 - o Statements in the structured block are executed in lexical iteration order
- OpenMP 4.5 **depend** clause for the **ordered** construct:
 - The **depend** clause is used to express dependences on earlier iterations via **sink** and **source** arguments
 - The **ordered** construct is placed within the loop body
 - Used to specify cross-iteration dependences

Doacross Concept

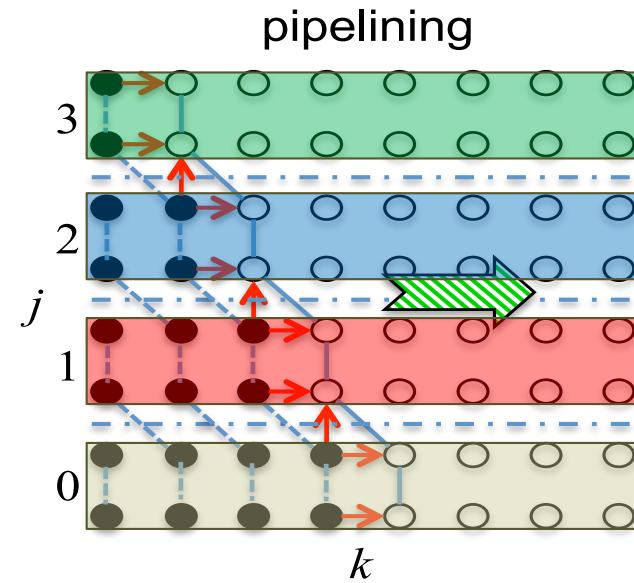
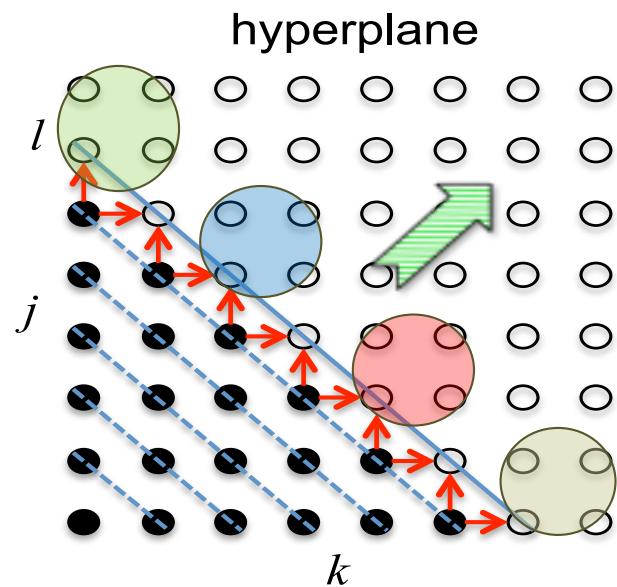
Wait point for completion
of iteration i-1

Signal completion of
iteration i

```
#pragma omp for ordered(1) {
    for (i=1; i<n; i++)
        ...
#pragma omp ordered depend(sink: i-1)
        b(i) = foe (a[i], b[i-1]);
#pragma omp ordered depend(source)
        ...
}
```

Exploiting Parallelism with Doacross

- Hyperplane or pipeline algorithms lend themselves to doacross parallelism
- Example: The LU NAS Parallel Benchmark
 - Symmetric Successive Over-relaxation (SSOR) algorithm to solve Navier-Stokes



The NPB LU Pseudo-Application



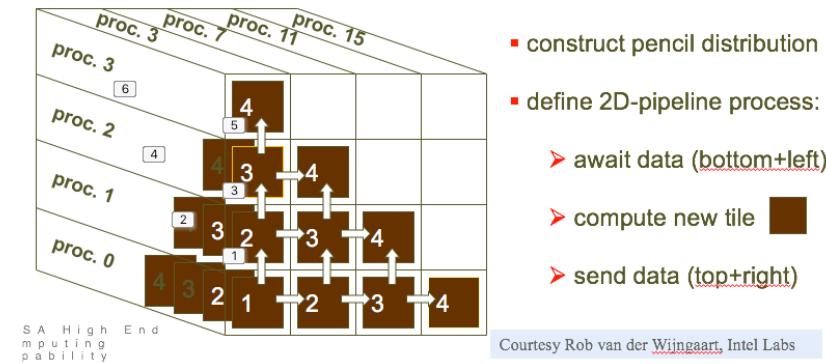
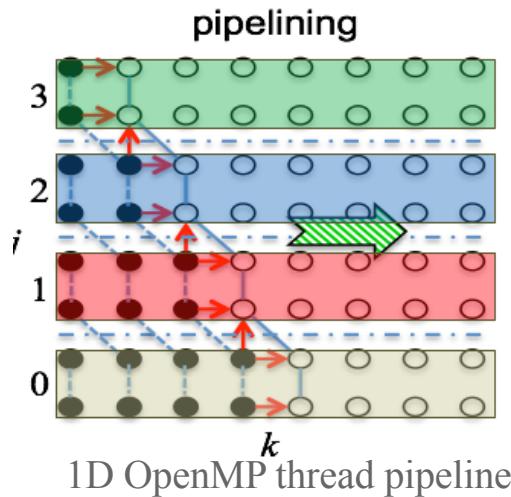
- The code employs SSOR to solve a 7-band, block-diagonal matrix
- The method is factorized into **Lower** and **Upper** solver steps
- Both solver steps carry dependences in each spatial dimension
 - prevents straightforward OpenMP parallelization

```
...
do k=kst, kend
    do j=jst, jend
        do i=ist, iend
            v(i,j,k)=v(i,j,k)+a*v(i-1,j,k)+b*v(i,j-1,k)
                + c*v(i,j,k-1) + d
        enddo; enddo; enddo
```

LU Manual Thread Pipelining



- Setting up a 1D OpenMP thread pipeline:
 - Place parallel construct on the outer loop
 - Place work sharing construct on the next inner loop
 - Synchronize with left and right neighbors in the inner loop
 - Synchronization relies on testing the value of shared variables
 - use OpenMP **flush** semantics
 - Manual implementation is time consuming, error prone, non-intuitive



Courtesy Rob van der Wijngaart, Intel Labs 6

Not possible in OpenMP



LU Thread Manual Pipelining vs Doacross

- Using OpenMP Doacross
 - Place the **parallel** construct on k loop
 - Place the workshare and the **ordered** clause on the j loop
 - Place **ordered** constructs with **depend** clause on the j loop body
 - The j-loop is now ordered according to the dependences specified

```
!$omp parallel manual
do k=kst, kend
    call sync_left(nx,ny,nz,rsd)
!$omp do schedule(static)
    do j=jst, jend
        call jacl(a,b,c,d,j,k)
        call blts(a,b,c,d,rsd,j,k)
    enddo
!$omp enddo nowait
    call sync_right(nx,ny,nz,rsd)
enddo
```

```
subroutine blts(a,b,c,d,rsd,j,k)
do i = ist, iend
    do m = 1, 5
        ...
        v(m,i,j,k) = v(m,i,j,k)
            -omega*(ldy(m,1,i)*v(m,i-1,j,k)
            + v(m,i,j-1,k) ... .
    end do
end do
```

```
!$omp parallel doac
do k=kst, kend
    !$omp do schedule(static) ordered(1)
        do j=jst, jend
            !$omp ordered depend(sink:j-1)
                call jacl(a,b,c,d,j,k)
                call blts(a,b,c,d,rsd,j,k)
            !$omp ordered depend(source)
        enddo
    !$omp enddo nowait
enddo
```

sync_left

sync_right

Where is i?

i-loop is hidden in jacl, blts
i-loop in jacl is vectorizable



LU 2D Hyperplane Manual Implementation

- Hyperplane \mathbf{l} contains all points where $\mathbf{l} = \mathbf{j} + \mathbf{k}$
- Points on a hyperplane can be computed independently
- Transform loop over (\mathbf{j}, \mathbf{k}) into a loop over $(\mathbf{l}, \mathbf{jk})$, where \mathbf{jk} iterates over the points within the hyperplane
- No explicit thread-to-thread synchronization is required
- Requires restructuring of the loop

```
!$omp parallel private (j,l,k)
do l=lst, lend
 !$omp do schedule(static)
 do jk =max(l-jend,jst), min(l-2,jend)
   j = jk
   k = l - jk
   call jacl(a,b,c,d,j,k)
   call blts(a,b,c,d,rsd,j,k)
 enddo
 !$omp enddo
enddo
```

Implicit barrier synchronization



LU 2D Hyperplane Doacross Implementation



- Maintains the original code structure
 - Extend dependences across 2 dimensions
 - Work sharing on **k** loop with **ordered (2)** clause
 - **ordered** construct with 2 **sinks**

```
!$omp do schedule(static,1) ordered(2)
do k=kst, kend
    do j=jst, jend
        !$omp ordered depend(sink:k-1,j) &
        !$omp&           depend(sink:k,j-1)
            call jaclD(a,b,c,d,j,k)
            call blts(a,b,c,d,rsd,j,k)
    !$omp ordered depend(source)
    enddo
enddo
```

- Best work balancing achieved using chunk size 1

- Only dependences are specified
- Work schedule dependent on compiler generated synchronization and runtime library

LU 3D Hyperplane Manual Implementation



- Exploit additional parallelism across multiple dimensions
 - Lift the **i** index out of the subroutine
 - Pre-compute the number of points np(l) in a hyperplane
 - Loop through the hyperplanes (**l**) and points within the hyperplane (**n**)
 - Convert pairs of (**l,n**) into indices (**i,j**)
 - Place workshare construct on loop over hyperplanes

```
!$omp parallel
do l=kst+jst+ist, kend+jend+iendo
  !$omp do schedule(static)
    do n=1, np(l)
      ... convert indices ...
      k = l - j - i
      call jacl(a,b,c,d,i,j,k)
      call blts(a,b,c,d,rsd,i,j,k)
    enddo
  !$omp enddo
enddo
```

```
convert indices

j = max(l-jend-iend,jst)
i = n
ns = min(l-j-2,iend) - max(l-j-iend,ist) + 1
do while (i > ns)
  i = i - ns
  j = j + 1
  ns = min(l-j-2,iend) - max(l-j-iend,ist) + 1
end do
i = i + max(l-j-iend,ist) - 1
k = l - j - i
```



LU 3D Hyperplane Doacross

- The nested loops run within the original loop bounds
- Place **ordered(3)** clause on the triply nested loop
- Place **ordered** clause with 3 **sinks** in the loop body

```
!$omp do schedule(static,1) ordered(3)
do k = 2, nz -1
    do j = jst, jend
        do i = ist, iend
            !$omp ordered depend(sink: k-1,j,i) depend(sink: k,j-1,i)
            !$omp& depend(sink: k,j,i-1)
                call jacl(... i, j, k)
                call blts( isiz1, isiz2, isiz3,
                           >           nx, ny,nz,comega,crsd,a,b,c,d,i,j,k)
            !$omp ordered depend(source)
                end do
            end do
        end do
```

Evaluation Environment



- Pleiades Super Computer and a KNL cluster at NASA Ames Research Center
- Intel Xeon Broadwell dual E5-2680v4 (2.4GHz) with 28 cores
- Intel Xeon Phi ™ with CPU 7230 (1.3 GHz) with 64 cores
- GNU gcc 7.1 : gfortran provides OpenMP 4.5 support
 - `-O3 -fopenmp -mavx2 -g` for Xeon Broadwell
 - `-O3 -mavx512f -fopenmp -g` on KNL
- Intel ifort version 2017.1.132
 - `-O3 -ipo -axCORE-AVX2 -qopenmp -g` for Xeon Broadwell
 - `-O3 -xmic-avx512 -ipo -g` for KNL
- Performance analysis:
 - op_scope:
 - Low overhead profiling tool for OpenMP and MPI; licensed software
 - Paraver 4.6:
 - Flexible performance analysis tool distributed by the Barcelona Supercomputer Center



Naming Convention

Implementation	Name
Pipeline manual	manual-pipe
Pipeline doacross	doac-pipe
2D Hyperplane manual	manual-hp2
2D Hyperplane doacross	doac-hp2
3D Hyperplane manual	manual-hp3
3D Hyperplane doacross	doac-hp3

Benchmark Sizes

- LU Class A
 - $nx=ny=nz=64$
- LU Class C
 - $nx=ny=nz=162$

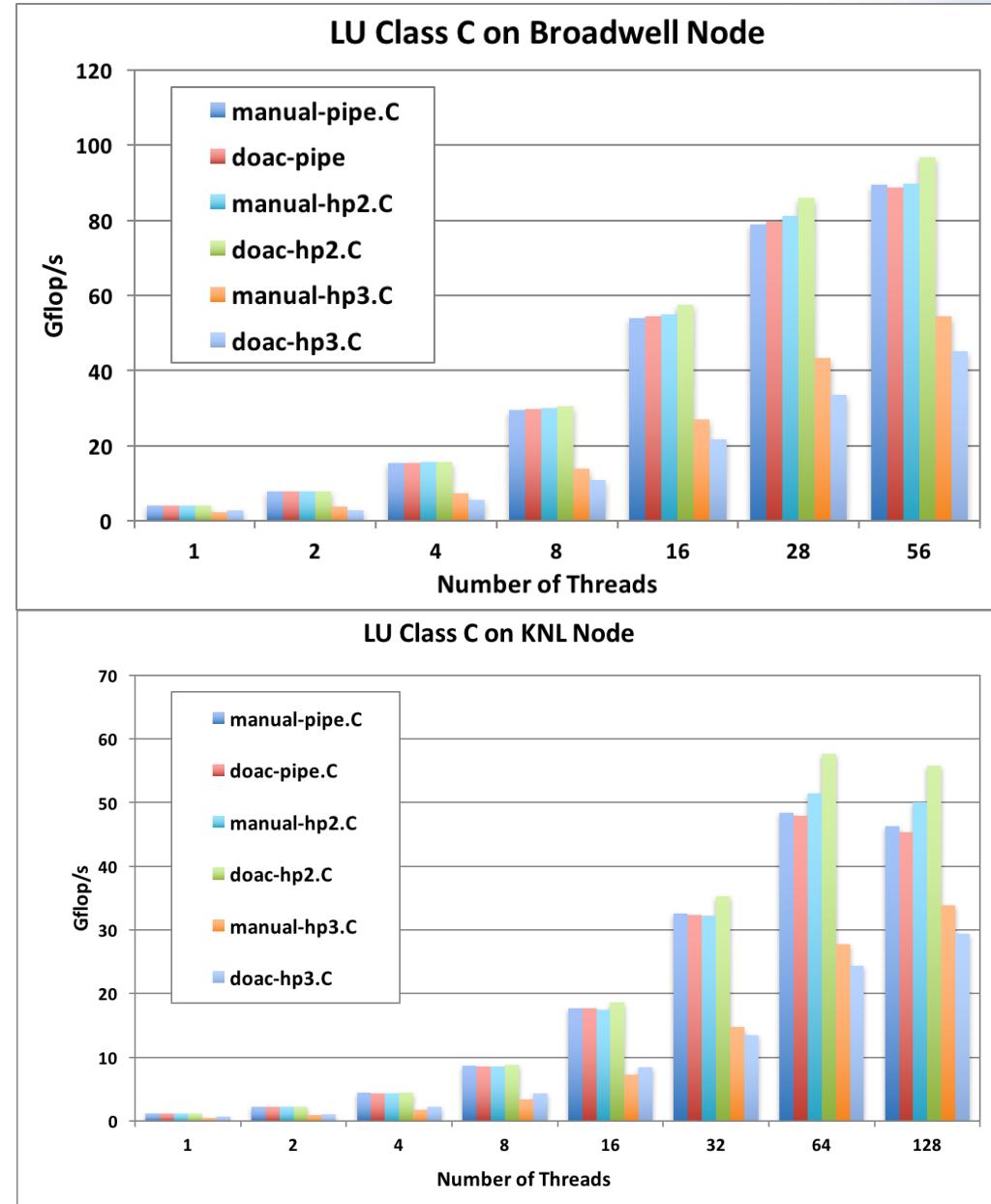
Thread binding

```
export OMP_PROC_BIND=spread
```

Class LU Class C Performance with GNU GCC



- Observations:
 - manual and doac performance is similar!
 - doac-hp2 slightly outperforms manual-hp2
 - Performance behavior on Broadwell and KNL is similar
 - hp2 performs better than pipelined execution
 - hp3 performs poorly



manual-pipe vs doac-pipe on Xeon Broadwell

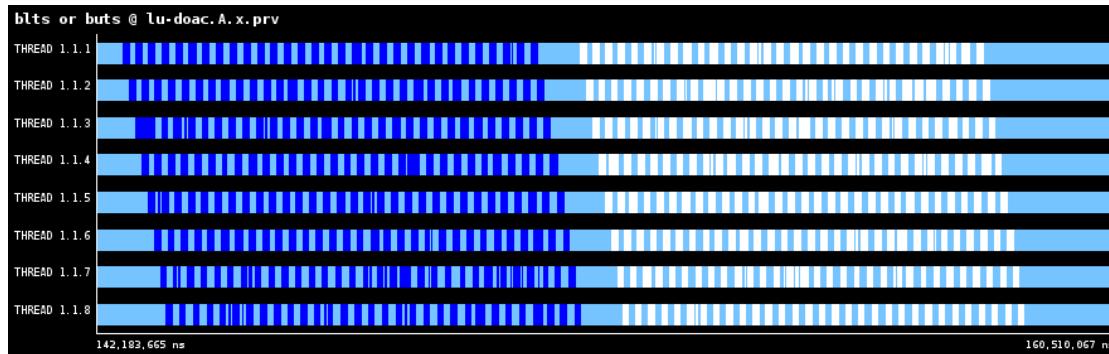


pipe-manual.C.x secs	doac-pipe.C.x secs
Elap: 30.71	Elap: 30.75
User: 767.40	User: 768.78
Sys: 67.48	Sys: 67.93
Thread: 19	Thread: 19
2.72 <-PARTIAL SUM	2.90 <-PARTIAL SUM
Total: 33.95 Symbol	Total: 33.88 Symbol
9.88 rhs_.omp_fn.0	9.83 rhs_.omp_fn
5.33 blts_	5.33 blts_
5.05 buts_	5.24 buts_
3.83 jacu_	3.67 jacld_
3.79 jacld_	3.48 jacu_
2.70 +sync_left_	2.89 +libgomp.so.1.0.0::GOMP_doacross_wait
1.88 ssor_.omp_fn.0	1.89 ssor_.omp_fn.0
1.35 libgomp.so.1.0.0::gomp_team_barrier_wait_end	1.3 libgomp.so.1.0.0::gomp_team_barrier_wait_end
0.07 exact_	0.07 exact_
0.03 erhs_.omp_fn.0	0.03 erhs_.omp_fn.0
0.02 +sync_right_	0.01 +libgomp.so.1.0.0::GOMP_doacross_post
0.01 libgomp.so.1.0.0::gomp_barrier_wait_end	0.01 libgomp.so.1.0.0::gomp_barrier_wait_end

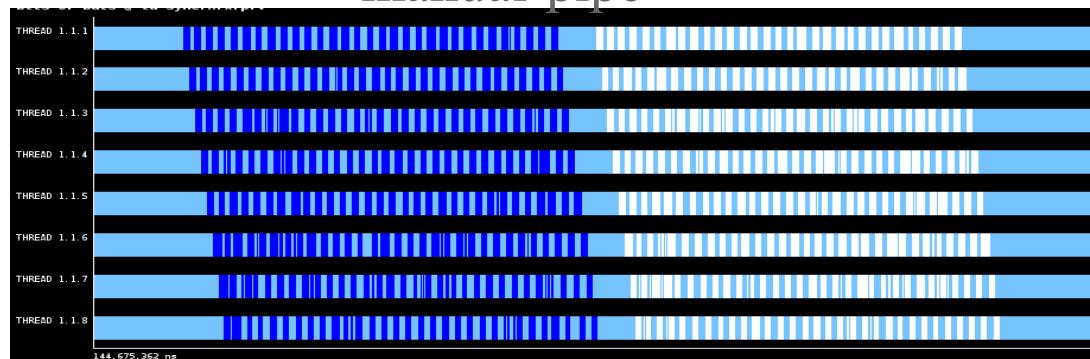
- *op_scope* profile for Class C; GNU GCC 7.1
- Time in seconds based on CPU_CLK_UNHALTED
- Displayed is the most time consuming thread
- Observation:
 - GOMP doacross synchronization time very close time in synchronization routines

Paraver Timelines for pipelined LU

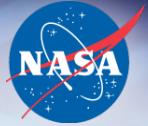
doac-pipe



manual-pipe



- Class A on 8 threads
- Paraver performance analysis tool time line
 - Vertical axis indicates time
 - Horizontal axis indicates thread ID
- Time spent in **blts** (dark blue), **buts** (white), **tracing disabled** (light blue)
- Traced are all even iterations
- Clearly detectable pipeline for both implementations



manual-hp2 vs doac-hp2 on Xeon Broadwell

manual-hp2.C.x

Elap: 30.56
User: 784.09
Sys: 66.56
Thread: 19
Event: CPU_CLK_UNHALTED
6.67 <-PARTIAL SUM

Total: 33.76 Symbol

9.75 lu-stat-hp2.C.x::rhs_omp_fn.0
6.67 +libgomp.so.1.0.0::gomp_team_barrier_wait_end
4.68 lu-stat-hp2.C.x::blts_
4.43 hp2.C.x::buts_
3.21 lu-stat-hp2.C.x::jacu_
3.08 lu-stat-hp2.C.x::jacld_
1.78 lu-stat-hp2.C.x::ssor_omp_fn.0
0.07 lu-stat-hp2.C.x::exact_

doac-hp2.C.x

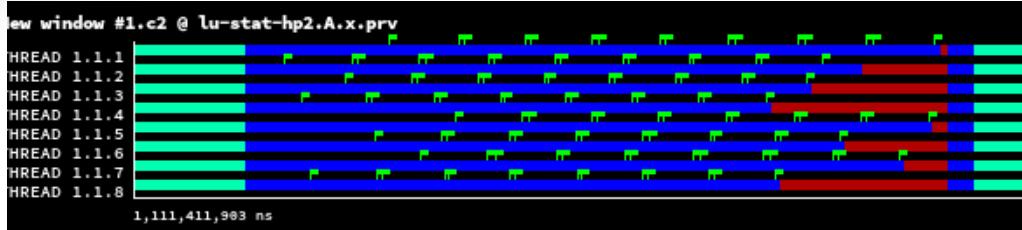
Elap: 28.48
User: 715.31
Sys: 61.55
Thread: 22
Event: CPU_CLK_UNHALTED
4.11 <-PARTIAL SUM

Total: 28.72 Symbol

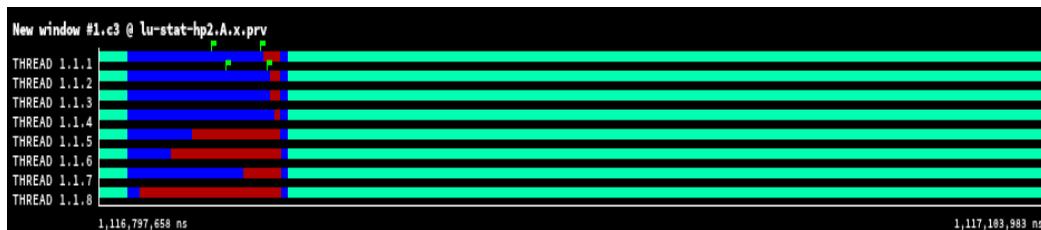
8.32 lu-doac-hp2.C.x::rhs_omp_fn.0
4.43 lu-doac-hp2.C.x::blts_
4.23 lu-doac-hp2.C.x::buts_
2.94 lu-doac-hp2.C.x::jacld_
2.85 lu-doac-hp2.C.x::jacu_
2.63 +libgomp.so.1.0.0::gomp_team_barrier_wait_end
1.73 lu-doac-hp2.C.x::ssor_omp_fn.0
1.46 +libgomp.so.1.0.0::GOMP_doacross_wait
0.02 +libgomp.so.1.0.0::GOMP_doacross_post

- op_scope profile for Class C on 28 threads
- Time in seconds based on CPU_CLK_UNHALTED
- Display thread with highest synchronization time
- PARTIAL SUM is the sum of all routines marked with “+”
- Observation:
 - Doacross implementation spends less time in synchronization than the manual hyperplane implementation

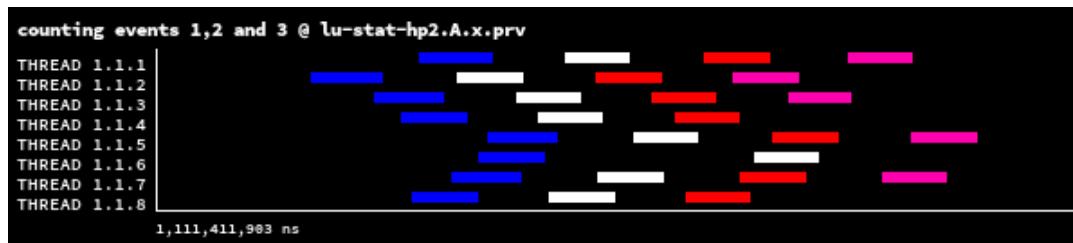
Hyperplane Workflow for manual-hp2



Running state for a hyperplane in the middle



Running state for a hyperplane towards the end



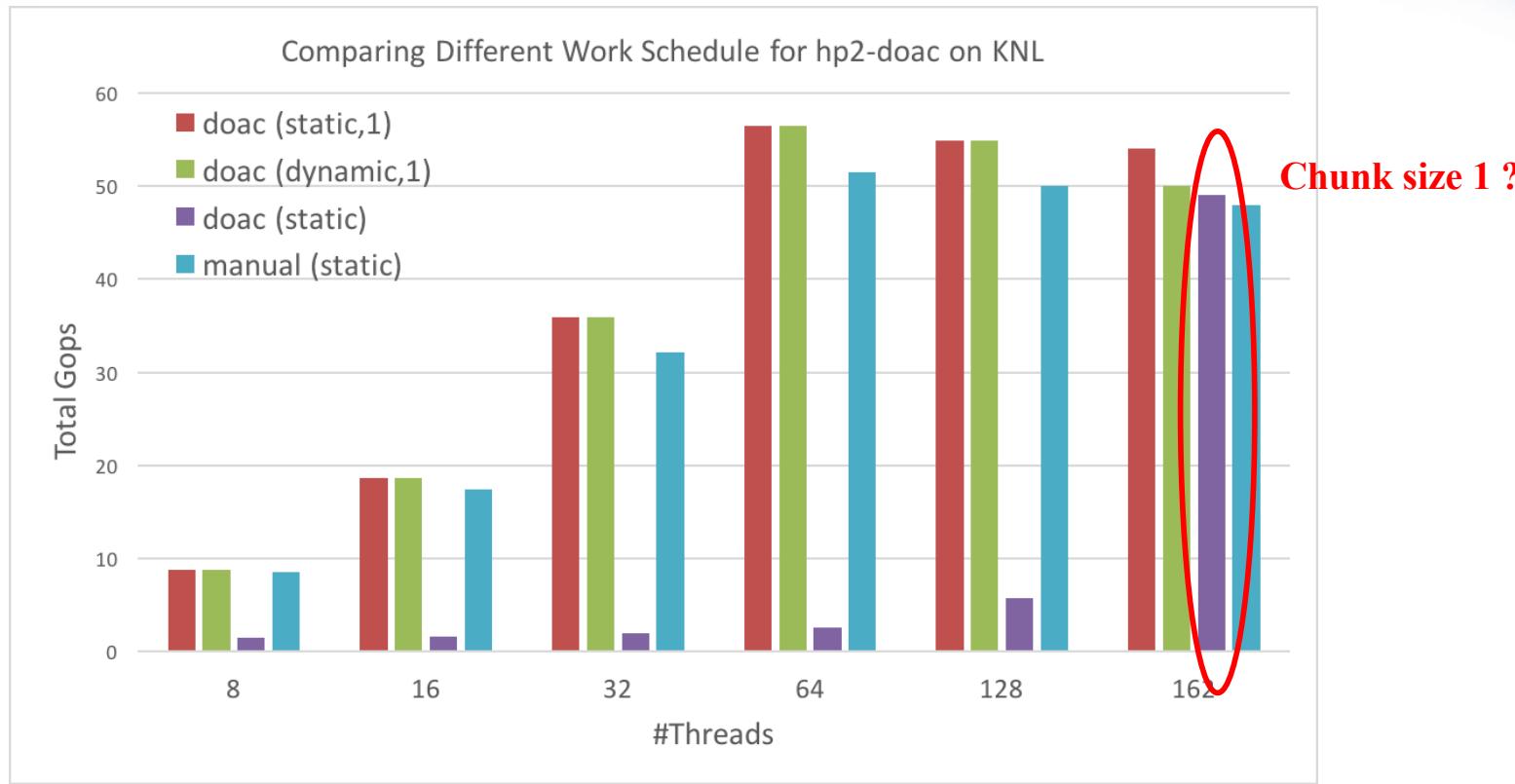
- Class A on 8 threads
- Paraver state timelines
 - **Running state** is dark blue
 - **Synchronization** is red
 - **Tracing disabled** is light green
- Observation:
 - More work in middle hyperplane
 - Higher load imbalance at end (and beginning) hyperplanes

Work flow for doac-hp2 implementations?
..... unknown,
synchronization is generated by the compiler

Middle hyperplane zoomed in: executions of loop body colored by chunk number



2D Hyperplane Work Schedule Comparison on KNL



- Class C, GNU gcc 7.1/gfortran
- Observations:
 - doac-hp2 with chunk size 1 performance is similar to manual-hp2
 - doac-hp2 static with chunk size > 1 performs poorly
 - Doac-hp2 static(schedule,1) performance boost for 162 threads !
- Similar observations on Xeon Broadwell

Class C Profiles for hp2-doac (static) on KNL



op_scope_ui.**doac-hp2-162**

Elap: 72.64
User: 6488.19
Sys: 4372.05
Thread: 110
Total: 46.94 Symbol

23.73 libgomp.so.1.0.0::GOMP_doacross_wait

7.18 libgomp.so.1.0.0::gomp_team_barrier_wait_end

5.41 lu-doac-hp2.C.x::rhs_.omp_fn.0
3.00 lu-doac-hp2.C.x::buts_
2.86 lu-doac-hp2.C.x::blts_
2.26 lu-doac-hp2.C.x::jacld_
1.97 lu-doac-hp2.C.x::jacu_

op_scope_ui.**doac-hp2-128**

Elap: 421.25
User: 43595.61
Sys: 3687.82
Thread: 87
Total: 413.89 Symbol

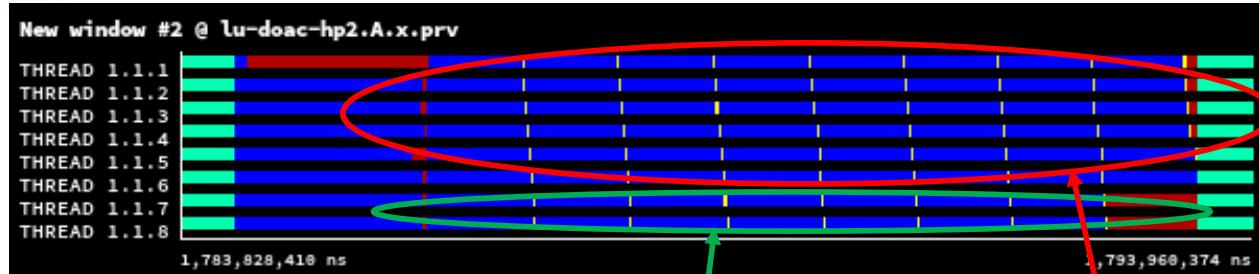
388.03 libgomp.so.1.0.0::GOMP_doacross_wait

8.36 libgomp.so.1.0.0:: omp_team_barrier_wait_end

5.37 lu-doac-hp2.C.x::rhs_.omp_fn.0
3.29 lu-doac-hp2.C.x::blts_
3.29 lu-doac-hp2.C.x::buts_
2.52 lu-doac-hp2.C.x::jacld_
2.51 lu-doac-hp2.C.x::jacu_

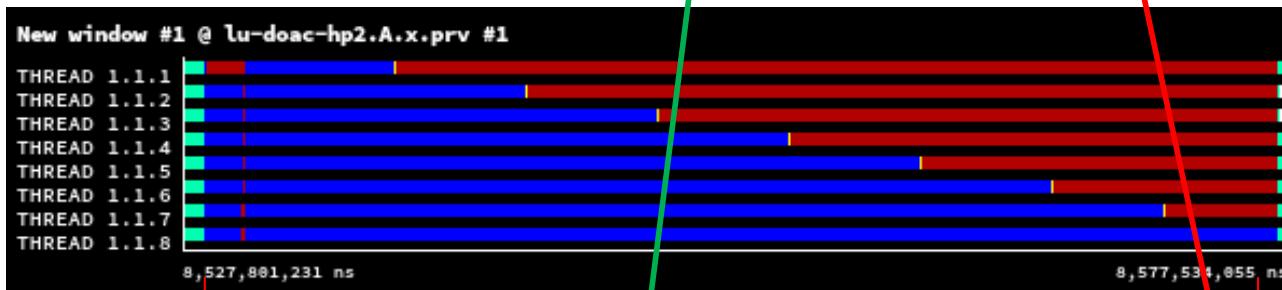
- Comparing runs on 162 (left) and 128 (right) threads
 - Displayed are the profiles for the thread taking most of the time
- Observations:
 - Excessive time in doacross synchronization for 128 threads
 - 162 threads achieve much better load balance than 128 threads
 - **Chunk size for the 162 run is 1 !?!**
- Similar observations of Broadwell

hp2-doac(schedule,1) vs (schedule) execution



hp2-doac schedule(static,1)

0.01 sec



hp2-doac schedule(static)

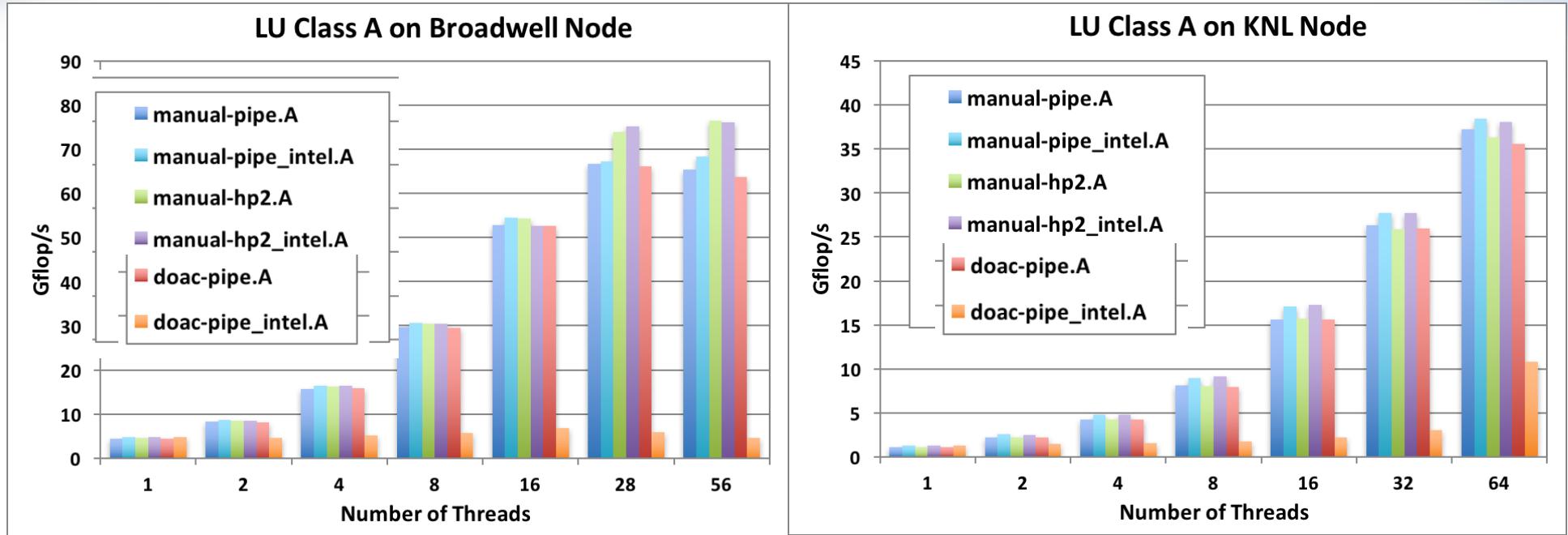
0.1 sec

7 chunks

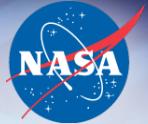
8 chunks

- Class A on 8 threads on Broadwell
- Paraver state time lines
 - State in lower solver step within one iteration
 - **Running** is blue, **Synchronizing** is red
 - Yellow tags indicate execution of chunks of the loop body
- Observations
 - **schedule (static,1)** yields balanced load
 - **schedule (static)** shows high load imbalance; each thread has 1 chunk of work, but chunks seem to be of unequal size

GNU gcc/gfortran vs Intel ifort Performance



- *version.A* indicates gcc performance, *version_intel.A* indicates ifort performance
- Observations:
 - gfortran and ifort yield similar performance for manual synchronization
 - Synchronization problem in ifort for doac-pipe implementations
- Similar Observations for Class C



doac gcc and ifort on Xeon Broadwell

manual-pipe.C.x secs

Elap: 30.71

User: 767.40

Sys: 67.48

Thread: 19

2.72 <PARTIAL SUM

Total: 33.95 Symbol

9.88 rhs_omp_fn.0

5.33 blts_

5.05 buts_

3.83 jacu_

3.79 jacld_

2.70 +sync_left_

1.88 ssor_omp_fn.0

1.35 libgomp.so.1.0.0::gomp_team_barrier_wait_end

0.07 exact_

0.03 erhs_omp_fn.0

0.02 +sync_right_

0.01 libgomp.so.1.0.0::gomp_barrier_wait_end

manual-pipe_intel.C.x secs

Elap: 554.15

User: 3574.32

Sys: 11920.94

Total: 105.44 Symbol

30.97 __kmpc_doacross_wait

28.98 __sched_yield

9.83 __kmp_yield

9.33 rhs_V

7.48 ssor_V

6.12 jacu_V

5.34 jacld_V

4.38 blts_V

1.60 sched_yield@plt

0.24 __kmp_hardware_timestamp

0.25 __kmpc_doacross_post

- *op_scope* profile for Class C on 28 threads
- Time in seconds based on CPU_CLK_UNHALTED
- Displayed is the most time consuming thread
- Observation:
 - Excessive synchronization time and system time for ifort generated code

Summary



- We compared different implementations of the NPB-OMP Benchmark LU
 - Manual synchronization vs
 - Synchronization via OpenMP 4.5 doacross
 - Using gcc 7.1
- OpenMP 4.5 provides ease of use
 - does not require restructuring of the original code
- Performance of manual synchronization vs OpenMP4.5 doacross is comparable if
 - There is sufficient data parallelism
 - doacross performance depends on choosing appropriate scheduling
 - o schedule (static,1) worked best for our LU benchmark
- Drawbacks:
 - Correctness and performance depends a lot on the quality of the compiler
 - gcc 7.1 provides full support (Fortran and C) and yields acceptable performance!
 - Performance and debugging is analysis difficult, due to the dependence on compiler transformation and OMP runtime library

References



- OpenMP Doacross:
 - Shirako J., Unnikrishnan P., Chatterjee S., Li K., Sarkar V. (2013) Expressing DOACROSS Loop Dependences in OpenMP. In: Rendell A.P., Chapman B.M., Müller M.S. (eds) OpenMP in the Era of Low Power Devices and Accelerators. IWOMP 2013. Lecture Notes in Computer Science, vol 8122. Springer, Berlin, Heidelberg
- OpenMP 4.5 Specifications
 - <http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf>
- OpenMP 4.5 Examples
 - <http://www.openmp.org/wp-content/uploads/openmp-examples-4.5.0.pdf>
- Paraver
 - <https://tools.bsc.es/paraver>
- op_scope
 - http://www.supersmith.com/site/op_scope_files/op_scope_3.0_draft.pdf
- NAS Parallel Benchmarks:
 - H. Jin, M. Frumkin, J. Yan, NAS Technical Report NAS-99-011 October 1999
 - <https://www.nas.nasa.gov/publications/npb.html>

References



- Overflow LU-SGS:
 - R.H.Nichols, R.W.Trmel. P.G.Buning, “Solver and Turbulence Model Upgrades to OVEFLOW 2 for Unstaedy and High-Speed Applications”, 25th Applied Aerodynamics Conference, 5-6 June 2006, San Francisco, California
- Parallel Research Kernels
 - <https://github.com/ParRes/Kernels>
- Pleiades Super Computer
 - <https://www.nas.nasa.gov/hecc/resources/pleiades.html>