OpenMP Case Studies

Dieter an Mey

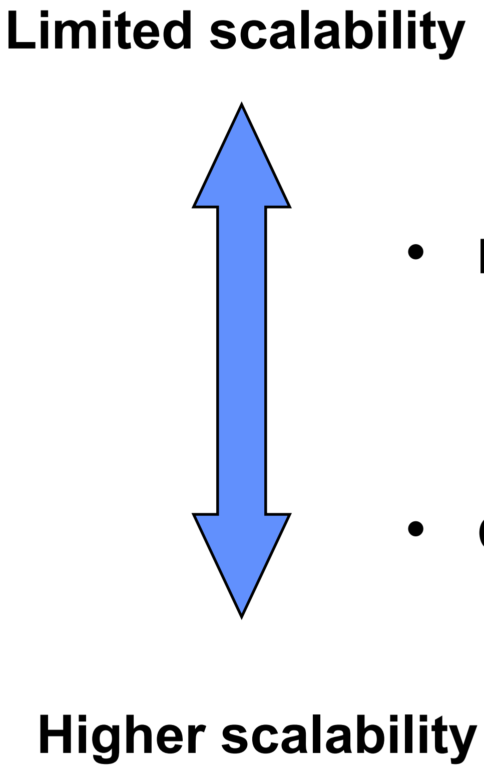
Center for Computing and Communication
Aachen University

anmey@rz.rwth-aachen.de

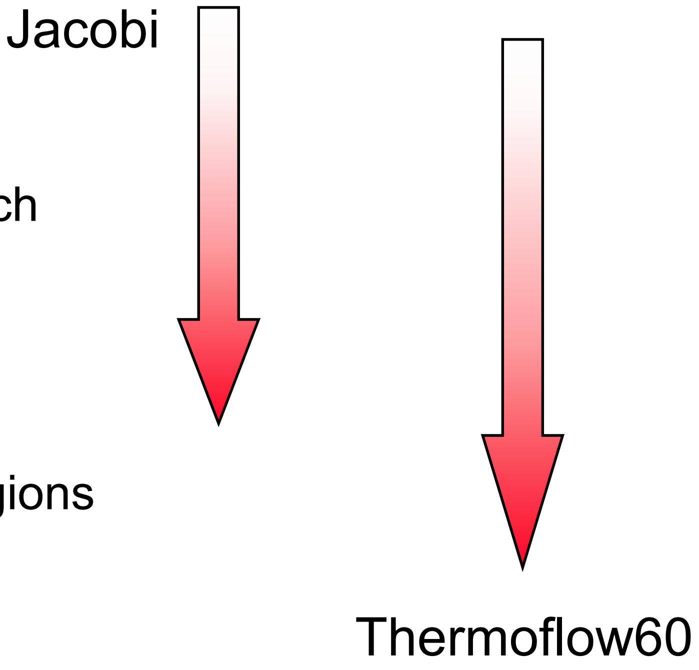
OpenMP Case Studies

- Parallelization Strategies
- A toy problem: The Jacobi method
- A real code: Thermoflow60 - FEM

Parallelization Strategies
Levels of OpenMP Parallelization



- Fine-grained parallelization
  - loop level
  - loop nest level
  - automatic parallelization
  - Easy to implement, stepwise approach
- Medium-grained parallelization
  - combining multiple parallel regions
  - Avoid barriers if possible
  - Orphaning, extracting the parallel regions
- Coarse-grained parallelization
  - Orphaning
  - Threadprivate
  - SPMD programming model
  - competes with MPI (but still needs shared memory)
- Hybrid parallelization with MPI and OpenMP



The Jacobi Example Program

www.openmp.org - Sample Programs

```
http://www.openmp.org/index.cgi?samples+samples/jacobi.html

************************************************************
* program to solve a finite difference
* discretization of Helmholtz equation :
* (d2/dx2)u + (d2/dy2)u - alpha u = f
* using Jacobi iterative method.
*
* Modified: Sanjiv Shah,       Kuck and Associates,Inc.(KAI),1998
* Author:   Joseph Robicheaux, Kuck and Associates,Inc.(KAI),1998
*
* Directives are used in this code to achieve paralleism.
* All do loops are parallized with default 'static' scheduling.
************************************************************
```

Jacobi Solver – Version 1
2 Parallel Regions

```
      error = 10.0 * tol
      k = 1
      do while (k.le.maxit .and. error.gt. tol)
         error = 0.0
         !$omp parallel do
            do j=1,m
               do i=1,n
                  uold(i,j) = u(i,j)
               enddo
            enddo
         !$omp end parallel do
         !$omp parallel do private(resid) reduction(+:error)
            do j = 2,m-1
               do i = 2,n-1
                  resid = (ax*(uold(i-1,j) + uold(i+1,j))
     &                    + ay*(uold(i,j-1) + uold(i,j+1))
     &                    + b * uold(i,j) - f(i,j))/b
                  u(i,j) = uold(i,j) - omega * resid
                  error = error + resid*resid
               end do
            enddo
         !$omp end parallel do
         k = k + 1
         error = sqrt(error)/dble(n*m)
      enddo
```

Autoparallelizing compilers typically generate an equivalent parallel code

Jacobi Solver – Version 1
2 Parallel Regions

```
      error = 10.0 * tol
      k = 1
      do while (k.le.maxit .and. error.gt. tol)
         error = 0.0
!$omp parallel do
            do j=1,m

               do i=1,n; uold(i,j) = u(i,j); enddo

            enddo
!$omp end parallel do
!$omp parallel do private(resid) reduction(+:error)
            do j = 2,m-1
               do i = 2,n-1

                  resid = (ax*(uold(i-1,j) ... )/b

                  u(i,j) = uold(i,j) - omega * resid
                  error = error + resid*resid
               end do
            enddo
!$omp end parallel do
         k = k + 1
         error = sqrt(error)/dble(n*m)
      enddo
```

This iteration loop is executed frequently!

FORK

JOIN

FORK

JOIN

Jacobi Solver – Version 2
only one Parallel Region

This version is distributed in www.openmp.org

```
      error = 10.0 * tol
      k = 1
      do while (k.le.maxit .and. error.gt. tol)
         error = 0.0
         !$omp parallel private(resid)
            !$omp do
               do j=1,m
                  do i=1,n; uold(i,j) = u(i,j); enddo
               enddo
            !$omp end do
            !$omp do reduction(+:error)
               do j = 2,m-1
                  do i = 2,n-1
                     resid = (ax*(uold(i-1,j) ... )/b
                     u(i,j) = uold(i,j) - omega * resid
                     error = error + resid*resid
                  end do
               enddo
            !$omp end do nowait
         !$omp end parallel
         k = k + 1
         error = sqrt(error)/dble(n*m)
      enddo
```

FORK

BARRIER

JOIN

Jacobi Solver – Version 3
Extracting the Parallel Region out of the Iteration Loop

```
      error = 10.0 * tol
      !$omp parallel private(resid,k_priv)                             FORK
         k_priv = 1
         do while (k_priv .le. maxit .and. error .gt. tol)
            !$omp do
               do j=1,m; do i=1,n; uold(i,j) = u(i,j); enddo; enddo
            !$omp end do                                               BARRIER
            !$omp single
               error = 0.0
            !$omp end single                                           BARRIER
            !$omp do reduction(+:error)
               do j = 2,m-1; do i = 2,n-1
                  resid = (ax*(uold(i-1,j) ... )/b
                  u(i,j) = uold(i,j) - omega * resid
                  error = error + resid*resid
               end do; enddo
            !$omp end do                                               BARRIER
            k_priv = k_priv + 1
            !$omp single
               error = sqrt(error)/dble(n*m)
            !$omp end single                                           BARRIER
         enddo
         !$omp single
            k = k_priv
         !$omp end single nowait                                       JOIN
      !$omp end parallel
```

Jacobi Solver – Version 3
Extracting the Parallel Region out of the Iteration Loop

```
    error = 10.0 * tol
    !$omp parallel pri
       k_priv = 1
       do while (k_priv .l        axit .and. error .gt. tol)
          !$omp do
             do j=1,m; do    =1,n; uold(i,j) = u(i,j); enddo; enddo
          !$omp end do
          !$omp single
             error = 0.0
          !$omp end single
          !$omp do reduction(+:err
             do j = 2,m-1; do i    2,n-1
                resid = (ax*(uold(i-1,j) ... )/b
                u(i,j) = uold(i,j) - omega * resid
                error = error + resid*resid
             end do; enddo
          !$omp end do
          k_priv = k_priv + 1
          !$omp single
             error = sqrt(error)/dble(n*m)
          !$omp end single
       enddo
       !$omp single
          k = k_priv
       !$omp end single nowait
    !$omp end parallel
```

`error` needs to be evaluated before it is set to 0

`uold` needs to be copied before it is used (overlap)

`error` needs to be written before the first thread updates it

the reduction result `(error)` is available after the next barrier

`error` needs to be calculated before it is used in the loop termination condition

FORK

BARRIER

BARRIER

BARRIER

ARRIER

Jacobi Solver – Version 4
Saving one Barrier in the Iteration Loop

```
!$omp parallel private(resid,k_priv,error_priv)
    k_priv = 1
    error_priv  = 10.0 * tol
    do while (k_priv .le. maxit .and. error_priv .gt. tol)
       !$omp do
          do j=1,m; do i=1,n; uold(i,j) = u(i,j)   enddo; enddo
       !$omp end do
       !$omp single
          error = 0.0
       !$omp end single
       !$omp do reduction(+:error)
          do j = 2,m-1; do i = 2,n-1
                   d = (ax*(uold(i-1
                   j) = uold(i,j) -
                   r = error + resic
                     enddo
       !$        nd do
       k_pr      = k_priv + 1
       error_    riv = sqrt(error)/dble(n*m)
    enddo
    !$omp barrier
    !$omp single
       k = k_priv
       error = error_priv
    !$omp end single nowait
!$omp end parallel
```

FORK

BARRIER

BARRIER

BARRIER

BARRIER

JOIN

the missing of this barrier has been detected by Assure

if the value of `error` is calculated redundantly by all threads, the single construct and its barrier is no longer needed

but then an additional barrier is necessary after the iteration loop, before a single thread provides the value of `error` in a shared variable

Jacobi Solver – Version 4
Saving one Barrier in the Iteration Loop

```
!$omp parallel private(resid,k_priv,error_priv)
    k_priv = 1
    error_priv  = 10.0 * tol
    do while (k_priv .le. maxit .and. error_priv .gt. tol)
       !$omp do
          do j=1,m; do i=1,n; uold(i,j) = u(i,j); enddo; enddo
       !$omp end do
       !$omp single
          error = 0.0
       !$omp end single
       !$omp do reduction(+:er
          do j = 2,m-1; do i = 2
             resid = (ax*(uold(i
             u(i,j) = uold(i,j)
             error = error + res
          end do; enddo
       !$omp end do
       k_priv = k_priv + 1
       error_priv = sqrt(error)/dble(n*m)
    enddo
    !$omp barrier
    !$omp single
       k = k_priv
       error = error_priv
    !$omp end single nowait
!$omp end parallel
```

The border values do not need to be copied (except for the first time)
=>
do j=2, m-1
is sufficient
=>
both parallel loops have the same limits

FORK

BARRIER

BARRIER

BARRIER

BARRIER

JOIN

Jacobi Solver – Version 5 (part 1)
No Worksharing Do Construct

```
nthreads = omp_get_max_threads()
jlo = 2
jhi = m-1
nrem = mod ( jhi - jlo + 1, nthreads )
nchunk = ( jhi - jlo + 1 - nrem ) / nthreads

!$omp parallel private(me,js,je,resid, k_local,error_local)

me = omp_get_thread_num()
if ( me  < nrem ) then
        js = jlo + me * ( nchunk + 1 )
        je = js + nchunk
else
        js = jlo + me * nchunk + nrem
        je = js + nchunk  - 1
end if

do while (k_priv .le. maxit .and. error_priv .gt. tol)
   ...
   do j=js,je; do i=1,n; uold(i,j) = u(i,j); enddo; enddo
   !$omp barrier
   ...
enddo
...
!$omp end parallel
```

the do directive is eliminated and precalculated loop limits are used

Jacobi Solver – Version 5 (part 2) No Worksharing Do Construct

```
!$omp parallel private(me,js,je,resid,k_priv,err_priv)
      ...
      k_priv = 1;  error_priv  = 10.0 * tol
      do while (k_priv .le. maxit .and. error_priv .gt. tol)
         do j=js,je; do i=1,n; uold(i,j) = u(i,j); enddo; enddo
         !$omp barrier
         !$omp single
            error = 0.0
         !$omp end single
         error_priv = 0.0
         do j = js,je; do i = 2,n-
               resid = (ax*(uold(          ... )/b
               u(i,j) = uold(i,j)       mega * resid
               error_priv = erro      iv + resid*resid
         end do; enddo
         !$omp critical
            error = error + er  r_priv
         !$omp end critical
         k_priv = k_priv + 1
         !$omp barrier
         error_priv = sqrt(error)/dble(n*
      enddo
!$omp single
         k = k_priv;  error = error_priv
!$omp end single nowait
!$omp end parallel
```

FORK

BARRIER

BARRIER

BARRIER

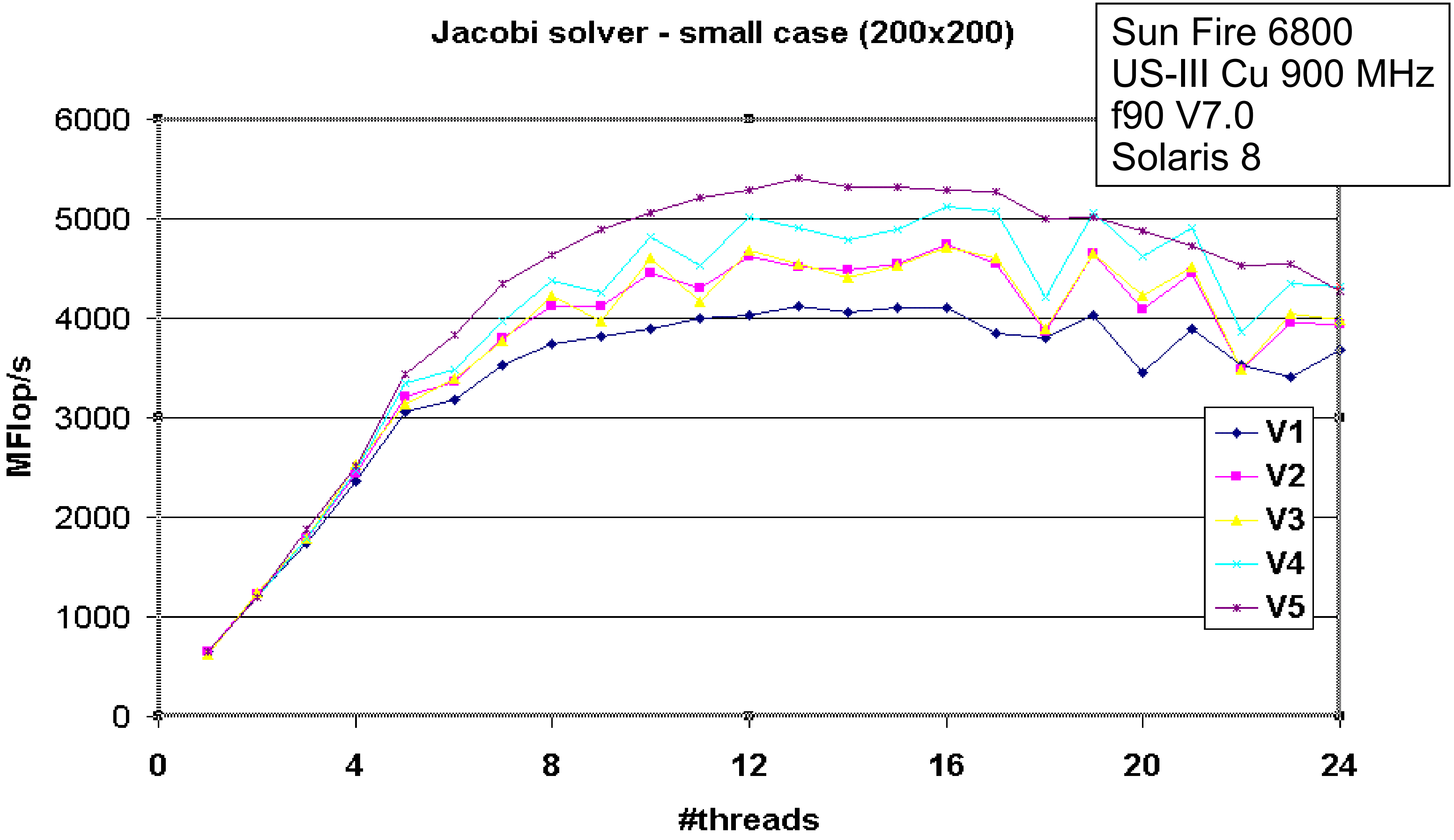
JOIN

the implicit barrier at the end do directive has to be replaced by an explicit barrier

the reduction construct has to be replaced by a critical section

This OpenMP coding style is no longer analyzable with Assure !

Jacobi Solver - Comparison



The ThermoFlow60 Finite-Element Program

Jet Propulsion Labatory Aachen University

Heat Flow Simulation with Finite Elements - ThermoFlow60

- simulation of the heat flow in a rocket combustion chamber
- 2D sufficient because of rotational symmetrie
- Finite Element method
- home-grown code
- 14 years of development
- has been vectorized before
- 29000 lines of Fortran
- ~ 200 OpenMP directives
- 69 parallel loops
- 1 main parallel region (orphaning)
- 200,000 cells
- 230 MB memory footprint
- 2 weeks serial runtime

First Approach to OpenMP

```
c$omp parallel do
c$omp&
c$omp&
C$omp&
C$omp&
          DO I=1,NEL
             K1     = IELM(I,1)
             K2     = IELM(I,2)
             K3     = IELM(I,3)
             DTEL   = .5d+00*DRI*(DTKN(K1)+DTKN(K2)+DTKN(K3))
             Q1     = U(K1)*DNDX(I,1)+V(K1)*DNDY(I,1)
             Q2     = U(K2)*DNDX(I,2)+V(K2)*DNDY(I,2)
             Q3     = U(K3)*DNDX(I,3)+V(K3)*DNDY(I,3)
             DTDRYE = DRI/YEL(I)
              ---  129 lines omitted ---
             q21 = tukl(i) * cq1 * cmy*dampqe*prode1*r  l(i)/epsl(i)
             q22 = tukl(i) * cq1 * prode2
             q23 = tukl(i) * cq1 * ( 1.0d+00+sark*xmatl )*ep  (i)/r  l(i)
             w21 = epsl(i) * cw1e * cmy*prode1*rhol(i)/epsl(i
             w22 = epsl(i) * cw1e * cw3*1.5d+00*prode2
             w23 = epsl(i) * cw2*epsl(i)/rhol(i)

             qtukl(i) = q21 + q22 - q23
             qepsl(i) = w21 + w22 - w23
          END DO
c$omp end parallel do
```

default(auto) ! **proposed to the OpenMP ARB**

default(__auto)
! Sun made a prototype implementation in the current Early Access Studio9 compiler !

loop over all elements

very error-prone
ASSURE helps!

many scalar, local temporary variables need to be privatized

All these arrays reside in (shared) COMMON blocks

Here Orphaning simplifies the Code ...

code outside of parallelized loops has to be put in single regions or has to be executed redundantly

use ASSURE to verify!

all the local variables are private by default

All these arrays in COMMON blocks remain shared

```
c$omp do


          DO I=1,NELM
             K1      = IELM(I,1)
             K2      = IELM(I,2)
             K3      = IELM(I,3)
             DTEL    = .5d+00*DRI*(DTKN(K1)+DTKN(K2)+DTKN(K3))
             Q1      = U(K1)*DNDX(I,1)+V(K1)*DNDY(I,1)
             Q2      = U(K2)*DNDX(I,2)+V(K2)*DNDY(I,2)
             Q3      = U(K3)*DNDX(I,3)+V(K3)*DNDY(I,3)
             DTDRYE = DRI/YEL(I)
              ---  129 lines omitted ---
             q21 = tukl(i) * cq1 * cmy*dampqe*prode1*rhol(i)/epsl(i)
             q22 = tukl(i) * cq1 * prode2
             q23 = tukl(i) * cq1 * ( 1.0d+00+sark*xmatl )*epsl(i)/rhol(i)
             w21 = epsl(i) * cw1e * cmy*prode1*rhol(i)/epsl(i)
             w22 = epsl(i) * cw1e * cw3*1.5d+00*prode2
             w23 = epsl(i) * cw2*epsl(i)/rhol(i)

             qtukl(i) = q21 + q22 - q23
             qepsl(i) = w21 + w22 - w23
          END DO
c$omp end do
```

Frequently used Loop Constructs

- Loop type 1, loop over (~100,000) FE nodes

```
!$omp do
do i = 1, npoin
   ...
end do
!$omp end do
```

- Loop type 2, loop over (~200,000) FE cells

```
!$omp do
do i = 1, nelm
   ...
end do
!$omp end do
```

- Loop (nest) type 3, loop over nodes and neighbours

```
!$omp do
do i = 1, npoin
   do j = 1, nknot(i)   ! varies between 3 and 6
      ...
   end do
end do
!$omp end do
```

Eliminating unnecessary Barriers

- Barriers between loops of the same type
- can in many cases be eliminated:

```
!$omp do
do i = 1, npoin
   ...
end do
!$omp end do nowait

!$omp do
do i = 1, npoin
   ...
end do
!$omp end do
```

Verify correctness with Assure !

Avoiding the Overhead of Worksharing Constructs

- Loop type 1, loop over (~100,000) FE nodes

```
    !$omp do
    do i = 1, npoin                   do i = ilo_poin, ihi_poin
       ...                     --->      ...
    end do                            end do
    !$omp end do                      !$omp barrier
```

- Loop type 2, loop over (~200,000) FE cells

```
    !$omp do
    do i = 1, nelm                    do i = ilo_elm, ihi_elm
       ...                     --->      ...
    end do                            end do
    !$omp end do                      !$omp barrier
```

- Loop (nest) type 3, loop over nodes and neighbours

```
    !$omp do
    do i = 1, npoin                   do i = ilo_knot, ihi_knot
       do j = 1, nknot(i)                do j = 1, nknot(i)
          ...                   --->        ...
       end do                            end do
    end do                            end do
    !$omp end do                      !$omp barrier
```

Precalculating the Loop Limits (1 of 2)

- **Loop type 1, loop over (~100,000) FE nodes**

```
      integer ilo_poin,ihi_poin,ilo_elm,ihi_elm,ilo_knot,ihi_knot
      common /omp_com/ilo_poin,ihi_poin,ilo_elm,ihi_elm,ilo_knot,...
!$omp threadprivate(/omp_com/)

      nrem_poin   = mod ( npoin, nthreads )   ! remaining nodes
      nchunk_poin = ( npoin - nrem_poin ) / nthreads   ! chunk size

!$omp parallel private(myid)
      myid = omp_get_thread_num()
      if ( myid < nrem_poin ) then
              ilo_poin = 1 + myid * ( nchunk_poin + 1 )
              ihi_poin = ilo_poin + nchunk_poin
      else
              ilo_poin = 1 + myid * nchunk_poin + nrem_poin
              ihi_poin = ilo_poin + nchunk_poin  - 1
      end if
!$omp end parallel
```

Even work distribution

- **Loop type 2, loop over (~200,000) FE cells**

```
    --- similar to loop type 1 ---
```

Precalculating the Loop Limits (2 of 2)

```
! Loop (nest) type 3, loop over n
      itotal = 0
      do i = 1, npoin
         itotal = itotal + nk
      end do
      nchunk_knot = itotal /

      itotal = 0
      ithread = 0
      ilo_temp(0) = 1
      do i = 1, npoin
         itotal = itotal + nk
         if ( itotal .ge. (it
            ihi_temp(ithread)
            ithread = ithread
            if ( ithread .ge.
            ilo_temp(ithread)
         end if
      end do
      ihi_temp(nthreads-1) = n
!$omp parallel private(myid)
      myid = omp_get_thread_nu
      ilo_knot = ilo_temp(myid
      ihi_knot = ihi_temp(myid
!$omp end parallel
```

Finding the optimal work distribution
for the i-loop just by counting

General applicable for constructs like

```
do i = 1, many
   do j = 1, func(i) ! few
      call same_amount_of_work(i,j)
   end do
end do
```

Alternative for a more general case:

precalculate (record) i_array and j_array
and the replay collapsed loop

```
do ij = 1, total
   i = i_array(ij)
   j = j_array(ij)
      call same_amount_of_work(i,j)
   end do
end do
```

Loop Nest with Precalculated Optimal Schedule

```
   do i = ilo_knot, ihi_knot
      do j = 1,nknot(i)
          ii    = iknot(i,j)     ! Element number
          kk    = iknel(i,j)     ! local node number (1-3)

      ---  28 lines omitted ---

      end do
   end do
c$omp barrier
```

This OpenMP coding style is no longer analyzable with Assure !

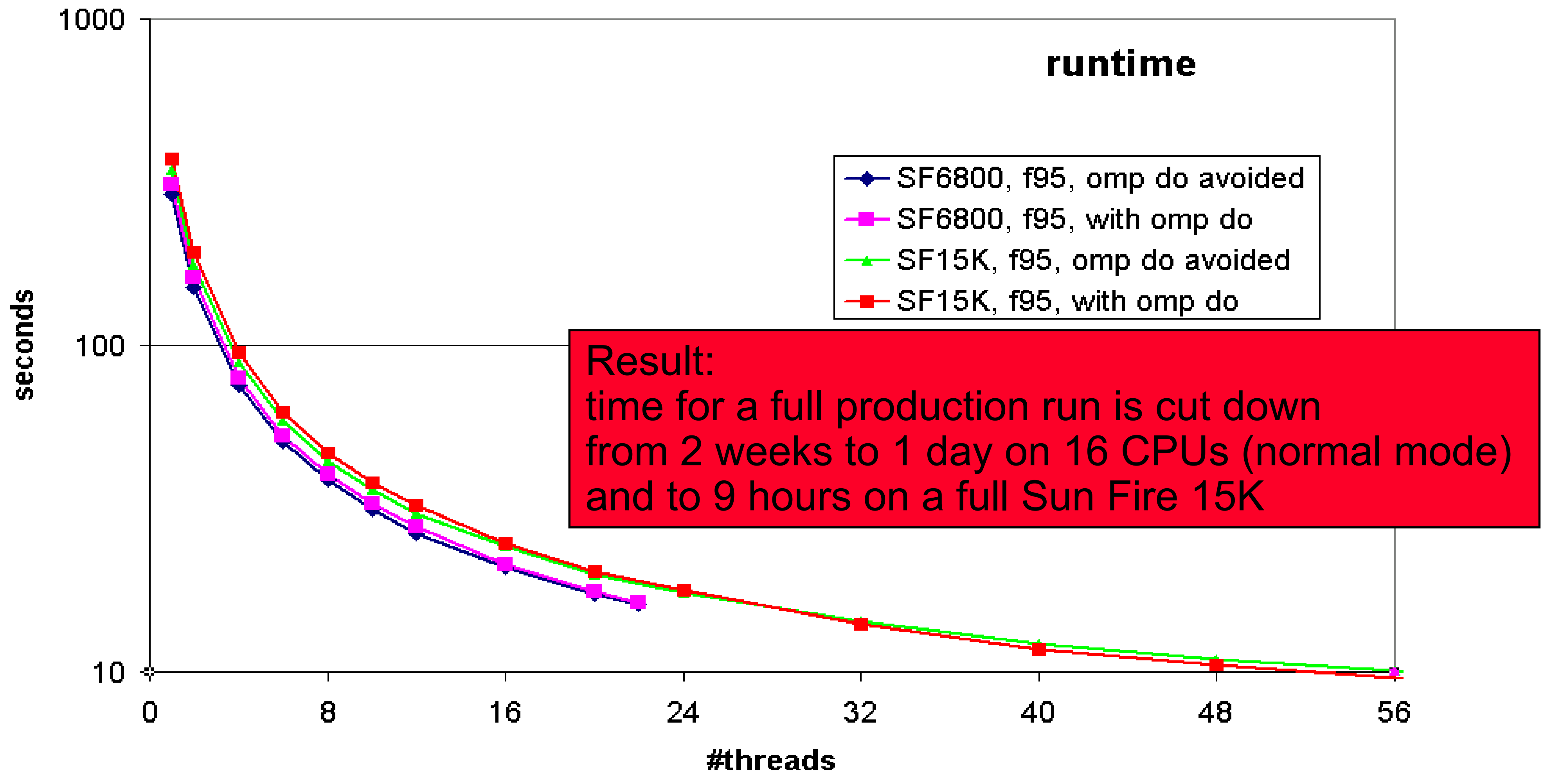
Machines used for Timing Measurements

- **Sun Fire 6800 with 24 CPUs**
  flat memory system with a bandwidth limited by the snooping bandwidth of 9,6 GB/s
- **Sun Fire 15K with 72 CPUs**
  cc-NUMA system with a backplane bandwidth of 43,2 GB/s
  snooping on board, directory-based cache coherency across boards

On both machines

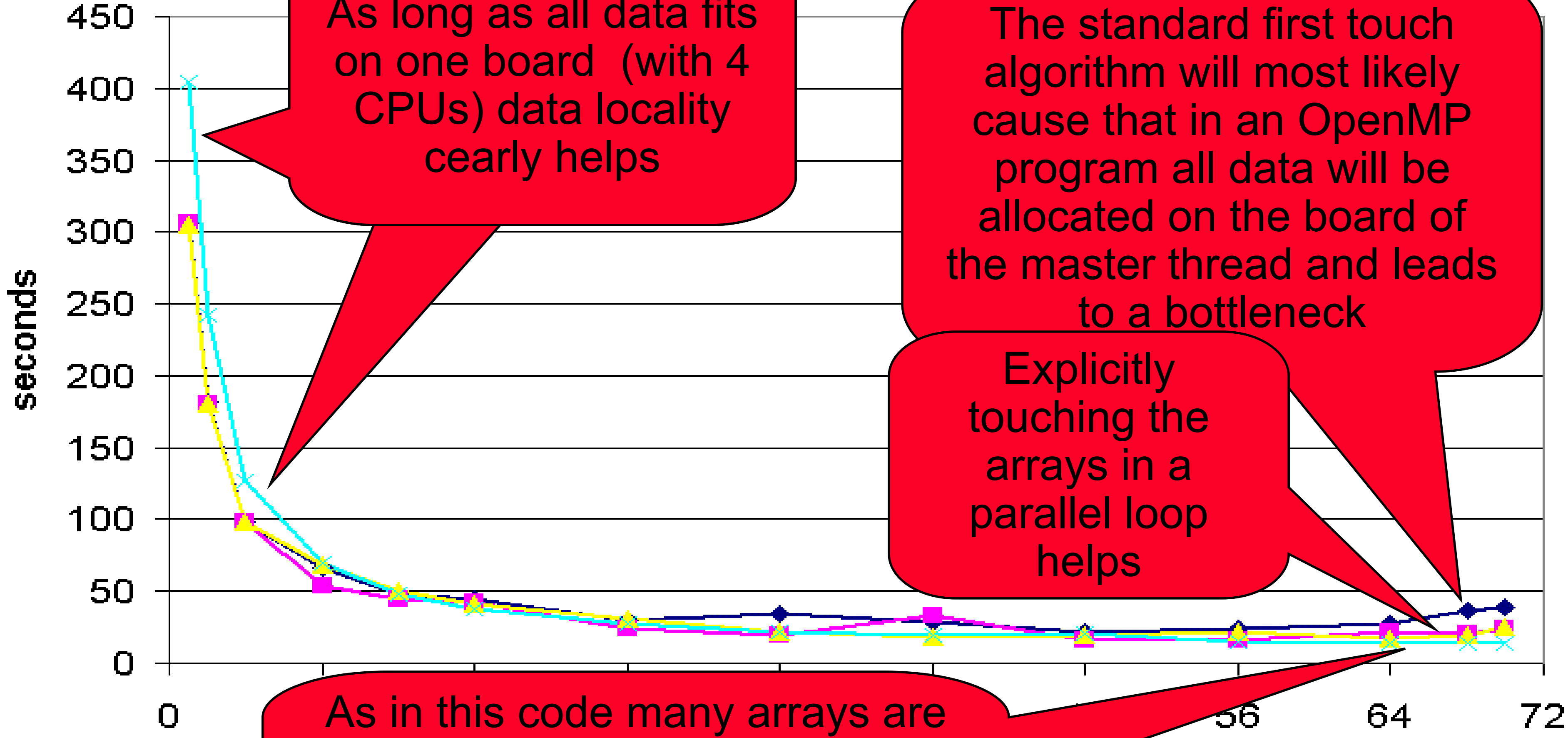
- 4 UltraSPARC-III Cu processors with 900 MHz clock cycles on one board with local memory
- CPU boards are connected together with a crossbar
- Solaris 8
  Locality of memory placement will not be supported before Solaris 9.1
- Sun ONE Studio 7 Fortran95 compiler
- KAP/Pro Toolset V 3.9 including the Guidef90 OpenMP preprocessor, which uses the native compiler as a backend

Comparison of Sun Fire 6800 and Sun Fire 15K (Solaris 8)



Result:
time for a full production run is cut down
from 2 weeks to 1 day on 16 CPUs (normal mode)
and to 9 hours on a full Sun Fire 15K

Solaris 9 versus Solaris 8 Memory Placement Optimization (MPO)



As long as all data fits on one board (with 4 CPUs) data locality cearly helps

The standard first touch algorithm will most likely cause that in an OpenMP program all data will be allocated on the board of the master thread and leads to a bottleneck

Explicitly touching the arrays in a parallel loop helps

As in this code many arrays are accessed sequentially and also indirectly, the random placement of Solaris 8 performs well for many threads

Summary

Summary

- It is possible to write a scalable OpenMP program with “only“ loop level parallelism.
- The parallel regions have to be extended as far as possible (Orphaning).
- Orphaning might even reduce the effort of variable scoping.
- The autoscoping feature of the upcoming Fortran compiler will be break-through for loop parallelization with OpenMP
- Avoid unnecessary barriers (still room for improvement)
- Verify the correctness of the OpenMP code with ASSURE
- Possibly replace OpenMP DO-constructs by precalculated DO-loop limits. It did not pay off here, but it might in a future version