OpenMP Case Studies

Dieter an Mey

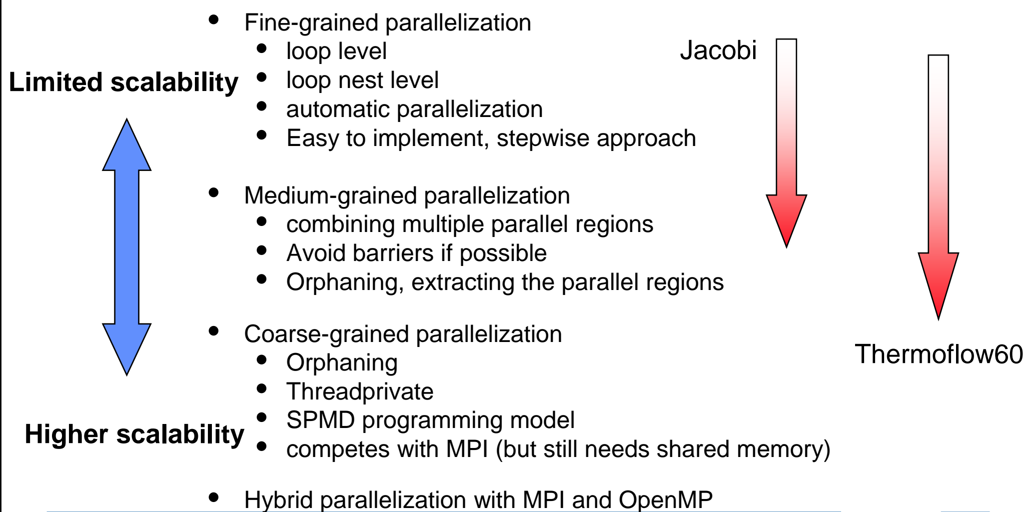
Center for Computing and Communication
RWTH Aachen University

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



The Jacobi Example Program

- Extending the parallel region
- Elimination of barriers
- Reductions and Software pipeling
- ccNUMA effects

www.openmp.org - Sample Programs

```
http://www.openmp.org/drupal/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.
************************************************************
```


Finite Difference Method

The 2D Helmholtz equation

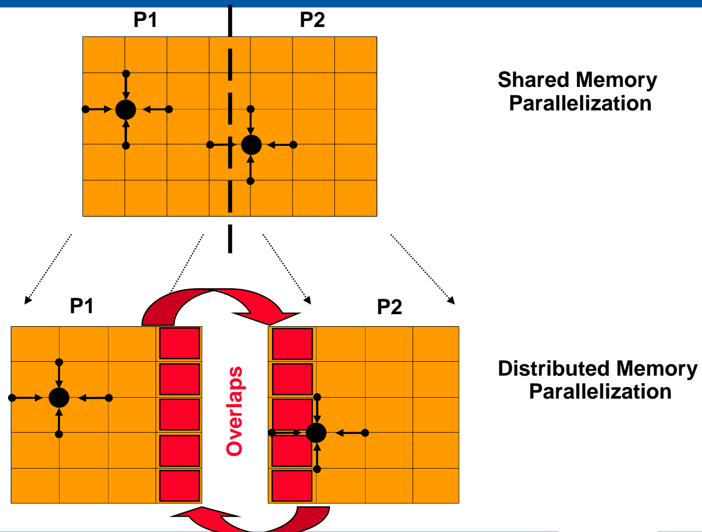
$$\Delta u + \kappa u = f$$

with homogeneous Dirichlet-boundary conditions is solved with a finite difference method.

The Laplace operator Δ is discretized with the central 5-point difference star . The corresponding lineare equations with a banded coefficient matrix are solved iteratively with the (simple) Jacobi method.

The Jacobi-method can easily be parallelized and also vektorized.

Jacobi-Method - Domain Decomposition



Jacobi Solver – Serial Version

```
error = 10.0 * tol
k = 1
do while (k.le.maxit .and. error.gt. tol)
   error = 0.0

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


      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

   k = k + 1
   error = sqrt(error)/dble(n*m)
enddo
```

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

This iteration loop is executed frequently!

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

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

      enddo                                            JOIN
   !$omp end parallel do
   !$omp parallel do private(resid) reduction(+:error) FORK
      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                               JOIN
   k = k + 1
   error = sqrt(error)/dble(n*m)
enddo
```

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

Taking a Closer Look at Reductions (1 of 5)

Standard case: OK

```
      error = 0.0
!$omp parallel
!$omp do reduction(+:error)
      do ..
         error = error + ..
      end do
!$omp end do
!$omp end parallel
      .. func(error) ..
```

```
 error = 0.0
!$omp parallel shared(error) private(temp)
  temp = 0
!$omp do
  do ..
     temp = temp + ..
   end do
!$omp end do
!$omp critical
   error = error + tmp
!$omp end critical
!$omp end parallel
   .. func ( error ) ..
```

This is one possible implementation of a reduction clause.

Taking a Closer Look at Reductions (2 of 5)

```
      error = 0.0
!$omp parallel
!$or
```

Standard case: OK

```
!$omp parallel
        ..
        error = 0.0
!$omp do reduction(+:error)
        do ..
            error = error + ..
        end do
!$omp end do
        .. func ( error ) ..
        ..
!$omp end parallel
```

Reduction variable has to initialized inside a larger parallel region:
No implied barrier at the beginning of a worksharing construct !

Data Race !

Wrong results!

Taking a Closer Look at Reductions (3 of 5)

```
      error = 0.0
!$omp parallel
!$or
```

Standard case: OK

```
!$omp parallel
        ..
```

Reduction variable has to initialized

```
!$omp parallel
        ..
!$omp master
        error = 0.0
!$omp end master
!$omp do reduction(+:error)
        do ..
            error = error + ..
        end do
!$omp end do
        .. func ( error ) ..
        ..
!$omp end parallel
```

Reduction variable is initialized in a master region (no implied barrier!):

Data Race !

Wrong results!

Taking a Closer Look at Reductions (4 of 5)

```
    error = 0.0
!$omp parallel
```

Standard case: OK

```
!$omp parallel
   ..
```

Reduction variable has to initialized

```
!$omp parallel
```

Reduction variable is initialized

```
!$omp parallel
      ..
!$omp single
      error = 0.0
!$omp end single
!$omp do reduction(+:error)
      do ..
         error = error + ..
      end do
!$omp end do
      .. func ( error ) ..
      ..
!$omp end parallel
```

Reduction variable is initialized in a single region (implied barrier!):

OK

Always correct results

Taking a Closer Look at Reductions (5 of 5)

```
    error = 0.0
!$omp parallel
```

Standard case: OK

```
!$omp parallel
   ..
```

Reduction variable has to initialized

```
!$omp parallel
```

Reduction variable is initialized

```
!$omp parallel
```

```
!$omp parallel
      ..
    error = 0.0
!$omp barrier
!$omp do reduction(+:error)
      do ..
         error = error + ..
      end do
!$omp end do
      .. func ( error )
      ..
!$omp end parallel
```

Reduction variable is initialized redundantly with an explicit barrier:

Data Race? Efficiency?

Correct results! Anyway: don't !

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
   !$omp parallel private(resid,k_priv)                          FORK
      k_priv = 1
      do while (k_priv .le. maxit .and. error .gt. tol)
         !$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
      !$omp end single nowait
   !$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

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

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


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

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


the missing of this barrier has been detected by a ThreadChecker

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

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

Jacobi Solver – Version 5 (1 of 2)
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 (2 of 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
         !$omp critical
            error = error + error_priv
         !$omp end critical
         k_priv = k_priv + 1
         !$omp barrier
         error_priv = sqrt(error)/dble(n*m)
      enddo
!$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

Jacobi Solver – Version 6 (1 of 2)
Extracting the Parallel Region out of the Routine

```
program driver
...
!$omp parallel
 call jacobi (n,m,alpha, &
        relax,u,f,tol,mits)
!$omp end parallel
...
end program driver
```

```
subroutine jacobi (n,m,alpha,omega,u,f,tol,maxit)
      integer n,m,maxit
      double precision dx,dy,f(n,m),u(n,m),alpha, tol,omega
      integer i,j,k,k_priv
      double precision error,resid,rsum,ax,ay,b
      double precision error_priv, uold(n,m)
      save error

      dx = 2.0 / (n-1)
      dy = 2.0 / (m-1)
      ax = 1.0/(dx*dx) ! X-direction coef
      ay = 1.0/(dy*dy) ! Y-direction coef
      b  = -2.0/(dx*dx)-2.0/(dy*dy) - alpha
!$omp single
      error = 10.0d0 * tol
!$omp end single
      k_priv = 1
      do while (k_priv.le.maxit .and. error.gt.tol)
        ...
      enddo                          ! End iteration loop
!$omp master
      print *, error
!$omp end master

      end subroutine jacobi
```

Jacobi Solver – Version 6 (2 of 2)
Extracting the Parallel Region out of the l

All these variables are still shared

All these variables are shared by default

```
program driver
...
!$omp parallel
 call jacobi (n,m,alpha, &
       relax,u,f,tol,mits)
!$omp end parallel
...
end program driver
```

```
subroutine jacobi (n,m,alpha,omega,u,f,tol,maxit)
      integer n,m,maxit
      double precision f(n,m),u(n,m),alpha,tol,omega
      integer i,j,k,k_priv
      double precision error,resid,rsum,ax,ay,b,dx,dy
      double precision error_priv, uold(n,m)
save error

      dx = 2.0 / (n-1)
      dy = 2.0 / (m-1)
      ax = 1.0/(dx*dx) ! X-direction coef
      ay = 1.0/(dy*dy) ! Y-direction coef
      b  = -2.0/(dx*dx)-2.0/(dy*dy) - alpha
!$omp single
      error = 10.0d0 * tol
!$omp end single
      k_priv = 1
      do while (k_priv.le.maxit .and. error.gt.tol)
         ...
      enddo                          ! End iteration loop
!$omp master
      print *, error
!$omp end master

      end subroutine jacobi
```

… unless they are declared to be static.

Local variables are private …

Jacobi Solver – Version 4 revisited (1 of 4)
Unrolling the while loop

```
!$omp parallel private(..)
     ..
        !$omp do
           .. uold.. = u ..
BARRIER !$omp end do
        !$omp single
        error = 0.0
BARRIER !$omp end single
        !$omp do reduction(+:error)
        do j,i..; u(i,j)=uold(i,j)..;error=error + ..; end do
BARRIER !$omp end do
        .. if ( f(error) ) exit
        !$omp do
           .. uold.. = u ..
BARRIER !$omp end do
        !$omp single
        error = 0.0
BARRIER !$omp end single
        !$omp do reduction(+:error)
        do j,i..; u(i,j)=uold(i,j)..;error=error + ..; end do
BARRIER !$omp end do
        .. if ( f(error) ) exit
     ..
!$omp end parallel
```

Jacobi Solver – Version 4 revisited (2 of 4) Software Pipelining

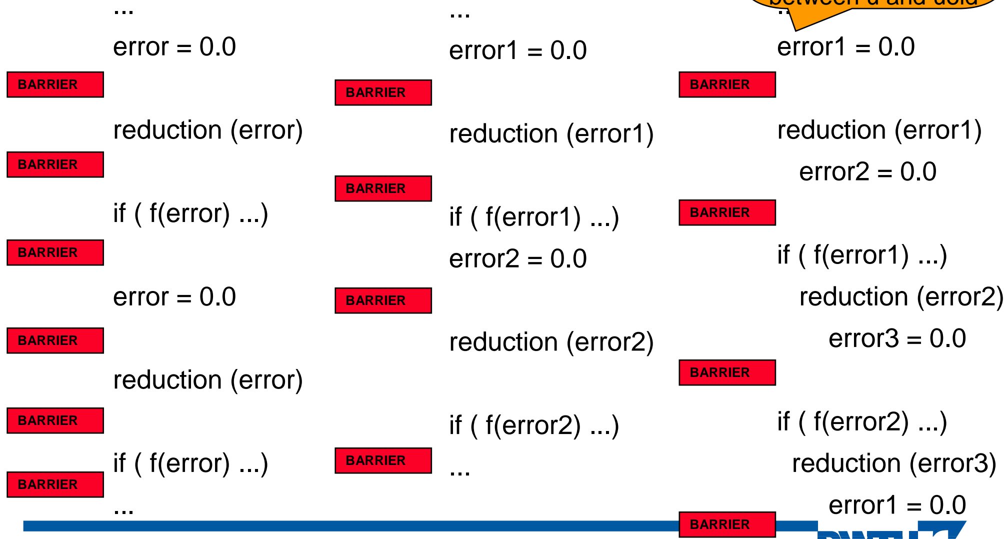
```
!$omp parallel private(..)
    ..
            !$omp do
               .. uold.. = u ..
BARRIER     !$omp end do
            !$omp single
            error1 = 0.0
BARRIER     !$omp end single
            !$omp do reduction(+:error)
            do j,i..; u(i,j)=uold(i,j)..;error1=error1 + ..; end do
BARRIER     !$omp end do
            .. if ( f(error1) ) exit
            !$omp do
               .. uold.. = u ..
BARRIER     !$omp end do
            !$omp single
            error2 = 0.0
BARRIER     !$omp end single
            !$omp do reduction(+:error)
            do j,i..; u(i,j)=uold(i,j)..;error2=error2 + ..; end do
BARRIER     !$omp end do
            .. if ( f(error2) ) exit
    ..
!$omp end parallel
```

Jacobi Solver – Version 4 revisited (3 of 4) Software Pipelining

```
!$omp parallel private(..)
    ..
            !$omp do
               .. uold.. = u ..
            !$omp end do nowait
            !$omp single
            error1 = 0.0
BARRIER     !$omp end single
            !$omp do reduction(+:error)
            do j,i..; u(i,j)=uold(i,j)..;error1=error1 + ..; end do
BARRIER     !$omp end do
            .. if ( f(error1) ) exit
            !$omp do
               .. uold.. = u ..
            !$omp end do nowait
            !$omp single
            error2 = 0.0
BARRIER     !$omp end single
            !$omp do reduction(+:error)
            do j,i..; u(i,j)=uold(i,j)..;error2=error2 + ..; end do
BARRIER     !$omp end do
            .. if ( f(error2) ) exit
    ..
!$omp end parallel
```

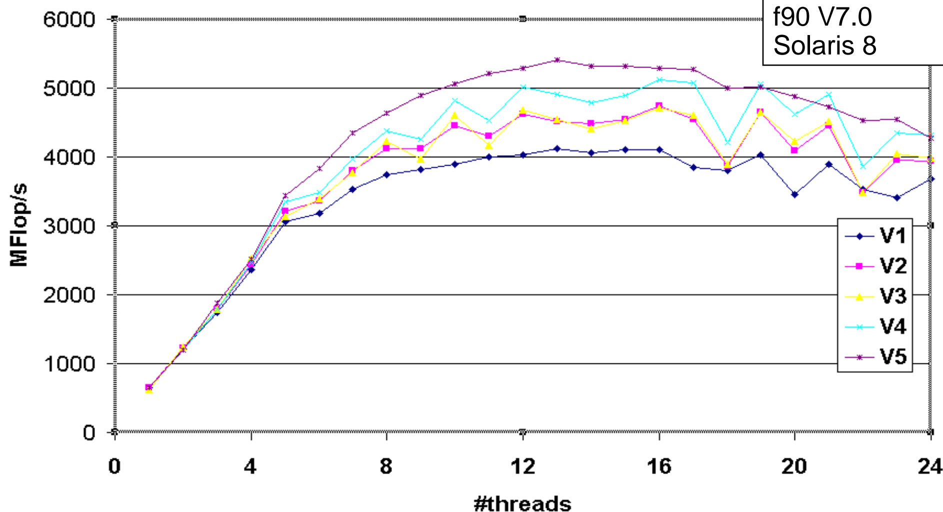
Jacobi Solver – Version 4 revisited
Software Pipelining

Only one barrier per iteration left ! Need to change algorithm: Swap between u and uold

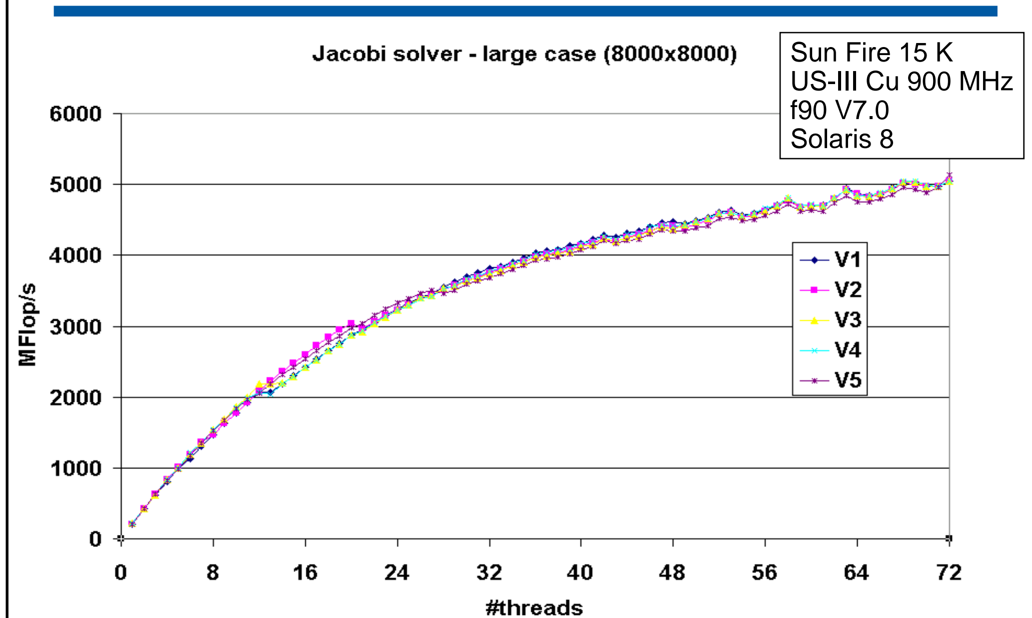


Jacobi Solver - Comparison

higher is better



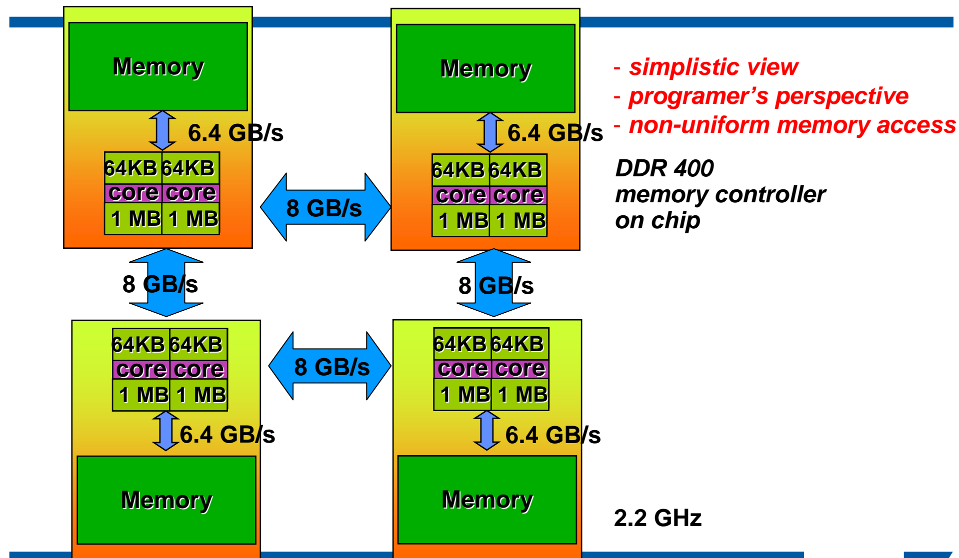
Jacobi Solver - Comparison



Dealing with ccNUMA Effects

- Some shared memory machines have physically distributed memory.
- Memory bandwidth and latency may vary depending on distance between thread and data.
- Try to allocate data close to where they are used.
- Data placement happens when data are initialized (and not allocated) (may be an issue when programming with C++ datatypes)
- Hopefully the operating system will not move threads around
  - Linux provides the taskset command to bind processes to sets of cpus
  - Sun Studio provides the **SUNW_PROCBIND** environment variable to bind threads
  - Enumerating the cores may depend on the OS

Sun Fire V40z at Aachen (can be used in the lab)



Revisiting Jacobi on a ccNUMA Machine

Call tree of the Jacobi example.

```
main (driver.F90)
     driver (driver.F90)
          initialize (driver.F90)  # the big arrays are initialized here
          jacobi (jacobi_omp_?.F90)  # has been discussed so far
          error_check (driver.F90)
```

Data Initialization in the Jacobi Example

- First Touch Policy:
  Arrays **u** and **f** are initialized using the same access pattern as later in the computation in subroutine **jacobi**.

```
 subroutine initialize (n,m,alpha,dx,dy,u,f)
 . . .
!$omp parallel do private(xx,yy,i,j)
      do j = 1,m
         do i = 1,n
            xx = -1.0 + dx * dble(i-1)        ! -1 < x < 1
            yy = -1.0 + dy * dble(j-1)        ! -1 < y < 1
            u(i,j) = 0.0
            f(i,j) = -alpha *(1.0-xx*xx)*(1.0-yy*yy) &
                 - 2.0*(1.0-xx*xx)-2.0*(1.0-yy*yy)
         enddo
      enddo
      return
      end
```


Data Migration with the Solaris Memory Placement Optimization feature (MPO)

- If data cannot be initialized in optimal manner, they can be explicitely migrated to where they are used next
- if data is accessed by many threads (e.g. random access), they can be scattered across all parts of the physical memory (Sun: locality groups)

```
/* Advise Solaris on how physical memory should be allocated for
* each array. */
if (madvise((caddr_t)first_touch_array, bytes, MADV_ACCESS_LWP)) {
       perror("madvise(MADV_ACCESS_LWP)");
       exit(1);
}
if (madvise((caddr_t)random_array, bytes, MADV_ACCESS_MANY)) {
       perror("madvise(MADV_ACCESS_MANY)");
       exit(1);
}
```

Performance Measurements on the Sun Fire V40z

Starting the Jacobi Example on the a 8-core Opteron-based machine with a 5000x5000 Matrix ~ 600 MB memory footprint

Initializing the matrix in a serial region causes two arrays out of three (`u` and `f`) to be allocated close to the master thread.

Before Migration the iterations take 0.423 seconds ~ 768 Mflop/s

Then all data are migrated to where they are used with `madvise()`.
Migration of some 40000 pages takes about 0.17 seconds

After Migration the iterations take about 0.094 seconds ~ 3457 Mflop/s
=> 4.5 x Improvement !

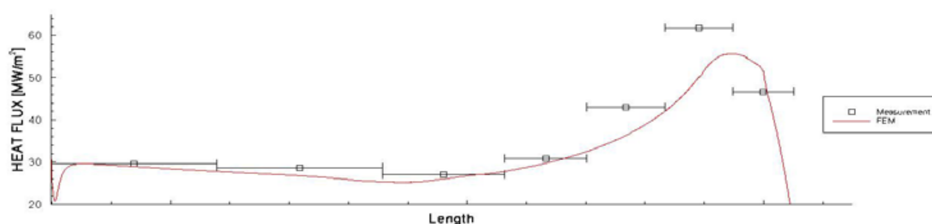
The ThermoFlow60 Finite-Element Program

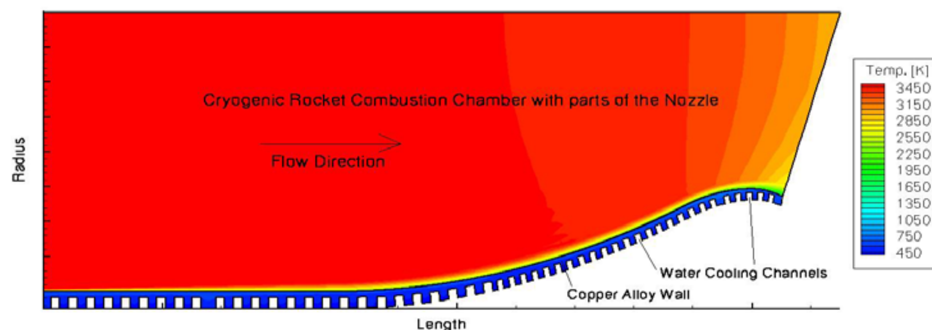
Jet Propulsion Labatory Aachen University

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

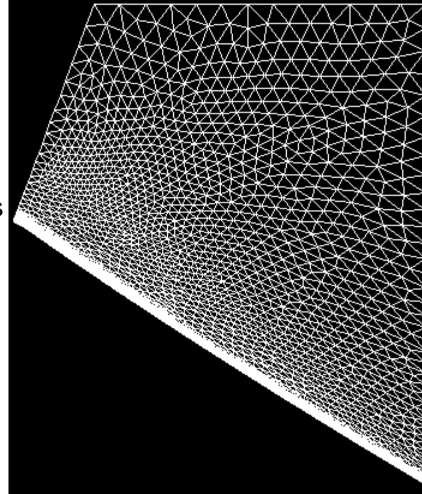
Simulation of the Heat Flow in a Rocket Combustion Chamber





The Grid

- very fine resolution at the boundary to the wall.
- no adaptation necessary
  =>
  loop limits remain constant
- coupled CFD- and structural analysis in preparation



First Approach to OpenMP

```
c$omp parallel do private(k1,k2,k3,dtel,q1,q2,q3,dtdrye,viture,ynorme,reture,
c$omp&           rhoe,tuke,epse,XMATL,PRODE1,prode2,dampqe,cw1e,q11,q12,
c$omp&           q13,w11,w12,w13,VILAML,VITURL,RLAMDA,RMUE2,DUDXNS,
C$omp&           DUDYNS,DVDXNS,DVDYNS,DTDX,DTDY,dq2dx,dq2dy,CPL,UUEL,
C$omp&           VVEL,VISK,VISE,q21,q22,q23,w21,w22,w23)
      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 parallel do
```

many scalar, local temporary variables need to be privatized

loop over all elements

very error-prone! A thread checking tools help!

All these arrays reside in (shared) COMMON blocks

First Approach to OpenMP

```
c$omp parallel do default(auto)          ! proposed to the OpenMP ARB
c$omp&            default(__auto)
c$omp&            ! Support by the Sun Studio compilers
C$omp&
C$omp&
      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) * cql * cmy*dampqe*prodel*rhol(i)/epsl(i)
         q22 = tukl(i) * cql * prode2
         q23 = tukl(i) * cql * ( 1.0d+00+sark*xmatl )*epsl(i)/rhol(i)
         w21 = epsl(i) * cwle * cmy*prodel*rhol(i)/epsl(i)
         w22 = epsl(i) * cwle * 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
```


Here Orphaning simplifies the Code ...

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

```
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) * cql * cmy*dampqe*prodel*rhol(i)/epsl(i)
         q22 = tukl(i) * cql * prode2
         q23 = tukl(i) * cql * ( 1.0d+00+sark*xmatl )*epsl(i)/rhol(i)
         w21 = epsl(i) * cwle * cmy*prodel*rhol(i)/epsl(i)
         w22 = epsl(i) * cwle * 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:

```
!$omp do schedule(static,chunksize)
do i = 1, npoin
   ...
end do
!$omp end do nowait

!$omp do schedule(static,chunksize)
do i = 1, npoin
   ...
end do
!
```

Verify correctness with a thread checking tool !

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

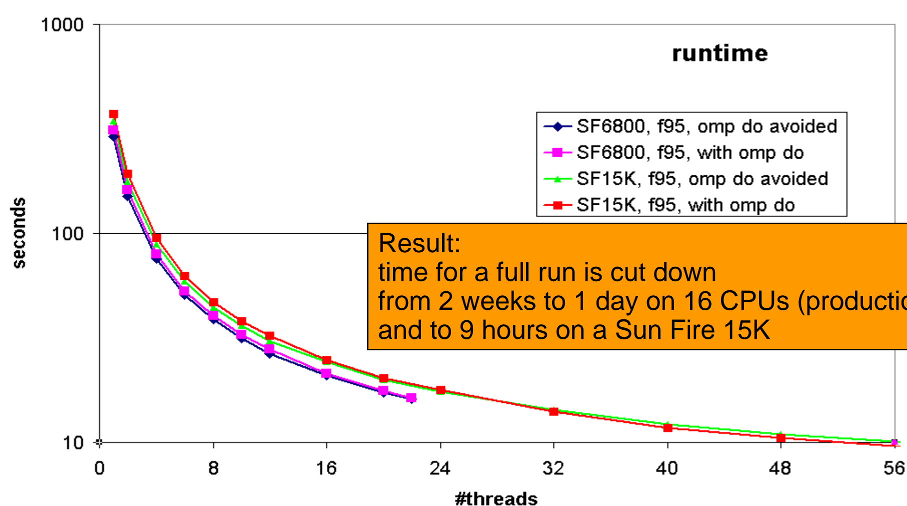
```
   do i = ilo_knot, ihi_knot
      do j = 1,nknot(i)
          ii    = iknot(i,j)     ! Element number
          kk    = iknel(i,j)     ! local node number (1-3)

      ---  28 lines omitted ---

      end do
   end do
c$omp barrier
```

Machines used for Timing Measurements

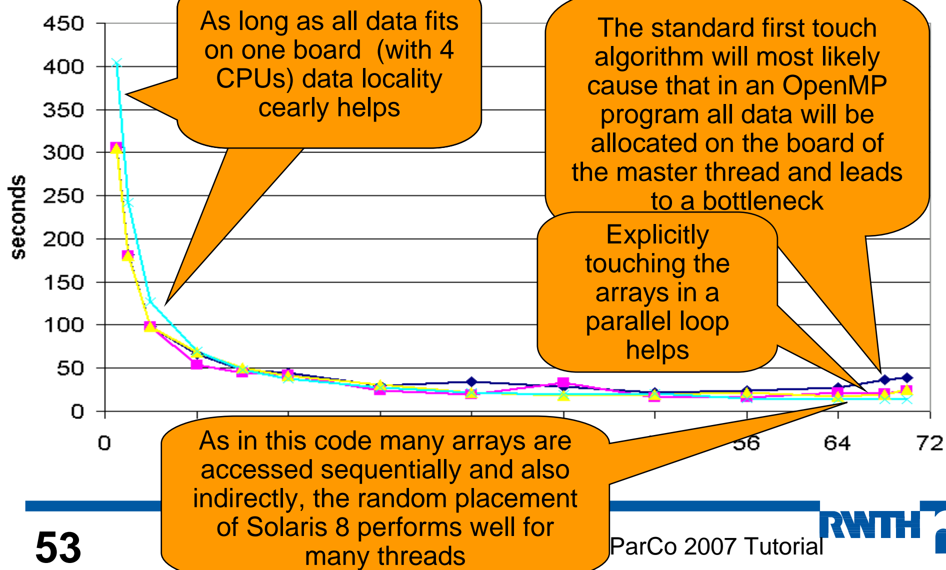
- **Sun Fire 6800 with 24 CPUs**
  flat memory system with a bandwidth limited by the snooping bandwidth of 9,6 GB/s
- **Sun Fire 15K with 72 CPUs**
  cc-NUMA system with a backplane bandwidth of 43,2 GB/s
  snooping on board, directory-based cache coherency across boards

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 (no ccNUMA awareness) versus
  Solaris 9.1 with Memory Placement Optimization (MPO)
- Sun ONE Studio 7 Fortran95 compiler

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



Solaris 9.1 versus Solaris 8 Memory Placement Optimization (MPO)



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).
- Variable scoping frequently is the hard part of OpenMP programming.
- Orphaning might even reduce the effort of variable scoping.
- The autoscoping feature of the Sun Studio compiler can be very handy.
- Avoid unnecessary barriers.
- Verify the correctness of the OpenMP code using thread checking tools.

References

- http://www.openmp.org – The OpenMP Architecture Review Board's (ARB) web site
- http://www.compunity.org – The OpenMP User Communities (cOMPunity) web site
- http://www.rz.rwth-aachen.de/sunhpc – The annual High Performance Computing on the Sun Fire SMP-Cluster Workshop
- Pushing Loop-Level Parallelization to the Limit - EWOMP'02, Rome, Italy, Sept 2002
- ParCo 2007 OpenMP Minisymposium