Pushing Loop-Level Parallelization to the Limit
Dieter an Mey¹, Thomas Haarmann², Wolfgang Koschel²

1) Center for Computing and Communication, Aachen University (RWTH) [1]
2) Jet Propulsion Laboratory, Aachen University (RWTH) [2]

1. Introduction

Good vectorizable code employs clean data structures and long loops with many operations and thus is also well suited for loop-level shared-memory parallelization with OpenMP. Nevertheless the scalability of such an approach is in many cases limited, because of the overhead of many parallel regions and worksharing constructs involved, unless measures are taken to overcome these limitations. In this paper, we first illustrate efficient parallelization techniques with a first simple example, and then show their impact in improving the scalability of a large Finite-Element (FE) program, used for the simulation of the heat flux in a rocket combustion chamber.

For the simple example and even more for the FE-code KAI’s ASSURE [3] turned out to be an indispensable tool for the development and verification of sophisticated OpenMP programs.

2. The Jacobi Example Program

The OpenMP Organization’s website [4] offers a nicely parallelized example program that solves the Helmholtz equation with a finite difference method on a regular 2D-mesh, using an iterative Jacobi method with over-relaxation. Although this might not be the optimal numerical method to solve such an equation, it is well suited for the study of various approaches of loop-level parallelization.

Inside the iteration loop there are two loop nests which could be automatically parallelized by all Fortran90 compilers we have used so far (HP, Sun, PGI, Visual KAP for OpenMP). The first code fragment (version 1, fig.1) shows the corresponding OpenMP version. The 2D-array uold is used to store the results of the previous iteration and the 2D-array u is used to store the results of the current iteration. In the first loop nest u is copied to uold and in the second loop nest the sweep operation is executed including the sum of the squared residuals used for the error estimation and the termination condition of the surrounding iteration loop.

```
subroutine jacobi (n,m,dx,dy,alpha,omega,u,f,tol,maxit)
implicit none
integer n,m,maxit
double precision dx,dy,f(n,m),u(n,m),alpha, tol,omega
integer i,j,k,k_local
double precision error,resid,rsum,ax,ay,b
double precision error_local, uold(n,m)
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 ! Central coeff
error = 10.0 * tol
k = 1
  do while (k.le.maxit .and. error.gt. tol) ! begin iteration loop
    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 ! end iteration loop
print *, 'Total Number of Iterations ', k
print *, 'Residual                   ', error
return
end

Fig. 1: Version 1 of the Jacobi solver with two parallel regions

The version which can be downloaded from the above mentioned website combines both parallel regions to reduce the parallelization overhead.

error = 10.0 * tol
k = 1

  do while (k.le.maxit .and. error.gt. tol) ! begin iteration loop
    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) + 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
    !$omp end do nowait
    !$omp end parallel
    k = k + 1
    error = sqrt(error)/dble(n*m)
  enddo ! end iteration loop

Fig. 2: Version 2 of the Jacobi solver with one parallel region containing both parallel loops (extract)
But the parallel region can even be further extended to contain the whole iteration loop (fig. 3). The code between the parallel do loops has to be executed redundantly or in single regions, which cause the introduction of two additional barriers at both `end single` directives.

```fortran
error = 10.0d0 * tol
!$omp parallel private(resid, k_priv)
  k_priv = 1
  do while (k_priv.le.maxit .and. error.gt.tol) ! begin iteration loop
    !$omp do
    do j=1,m
      do i=1,n
        uold(i,j) = u(i,j)
      enddo
    enddo
    !$omp end do
    !$omp single
    error = 0.0d0
    !$omp end single
    !$omp do 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
      enddo
    enddo
    !$omp end do
    k_priv = k_priv + 1
  !$omp end single
  error = sqrt(error)/dble(n*m)
  !$omp end single
enddo ! end iteration loop
!$omp single
  k = k_priv
!$omp end single nowait
!$omp end parallel
```

Fig. 3: Version 3 of the Jacobi solver with one parallel region containing the whole iteration loop (extract)

A careful investigation of the necessity of the barriers inside the iteration loop reveals that by replacing the shared variable `error` by a private copy `error_priv` in the termination condition of the iteration loop one out of four barriers can be eliminated (fig. 4)

```fortran
!$omp parallel private(resid, k_priv,error_priv)
  k_priv = 1
  error_priv = 10.0d0 * tol
  do while (k_priv.le.maxit .and. error_priv.gt.tol) ! begin iter. loop
    !$omp do
    do j=1,m
      do i=1,n
        uold(i,j) = u(i,j)
      enddo
    enddo
    !$omp end do
    !$omp single
    error = 0.0d0
    !$omp end single
    !$omp do 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
      enddo
    enddo
    !$omp end do
    k_priv = k_priv + 1
  !$omp end single
  error = sqrt(error)/dble(n*m)
  !$omp end single
enddo ! end iteration loop
!$omp single
  k = k_priv
!$omp end single nowait
!$omp end parallel
```
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
endd

!$omp end do

k_priv = k_priv + 1
error_priv = sqrt(error)/dble(n*m)
enddo ! end iteration loop

!$omp barrier
! by mistake this barrier was initially missing, detected by Assure

!$omp single
   k = k_priv
   error = error_priv
!$omp end single
!$omp end parallel

Fig. 4: Version 4 of the Jacobi solver with one parallel region containing the whole iteration loop one barrier avoided (extract)

The code in worksharing constructs are typically outlined into separate routines by an OpenMP compiler, which produces additional overhead. Also the splitting of the loop range into chunks causes additional index calculations which may slightly slow down the OpenMP version of a program.

After some minor program modifications, however, we could achieve that the two workshared do loops inside the iteration loop of the Jacobi program always have the same iteration space. So the worksharing directives can be easily eliminated and the loop chunks precalculated in front of the iteration loop (fig. 5). But additional barriers have to be inserted where beforehand the end do directives synchronized the threads. Also the reduction clause has to be replaced by a summation of private partial sums in a critical or atomic region.

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

! all parallel loops run from 2 to m-1

nthreads = omp_get_max_threads()
ilo = 2
ihi = m-1
nrem = mod ( ihi - ilo + 1, nthreads )
ncunk = ( ihi - ilo + 1 - nrem ) / nthreads

!$omp parallel private(me, is, ie, resid, k_priv, error_priv)

   me = omp_get_thread_num()
   if ( me < nrem ) then
      is = ilo + me * ( nchunk + 1 )
      ie = is + nchunk
   else
      is = ilo + me * nchunk + nrem
      ie = is + nchunk - 1
   end if
error_priv = 10.0 * tol
k_priv = 1

!$omp end parallel
do while (k_priv.le.maxit .and. error_priv.gt.tolh) ! begin iter. loop
  do j=is,ie
    do i=2,n-1
      uold(i,j) = u(i,j)
    enddo
  enddo
  !$omp barrier
  !$omp single
  error = 0
  !$omp end single
  error_priv = 0
  do j = is,ie
    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))*binv
      u(i,j) = uold(i,j) - omega * resid
      error_priv = error_priv + resid*resid
    end do
  enddo
  !$omp atomic
  error = error + error_priv
  k_priv = k_priv + 1
  !$omp barrier
  error_priv = sqrt ( error ) / dble(n*m)
enddo ! end iteration loop
 !$omp single
 k = k_priv
 !$omp end single
 !$omp end parallel
 error = sqrt ( error ) / dble(n*m)

Fig. 5: Version 5 of the Jacobi solver with one parallel region containing the whole iteration loop one barrier avoided and the worksharing constructs replaced (extract)

These 5 program versions have been measured on a Sun Fire 6800 system with 24 UltraSPARC III/Cu processors with a 900 MHz clock cycle (fig. 6). The Sun ONE Studio 7 compiler (formerly Forte Developer) has been used on a Solaris 8 operating system. All data for a grid size of 200 times 200 already fit into a single L2 cache, so the effect of a super linear speed-up was avoided. Of course the problem is too small to scale to a large number of processors. But it is small enough to do many experiments, and it is large enough to show the benefit of the described parallelization techniques. Choosing a large grid size the runtime of all versions is dominated by the memory bottleneck, such that the differences between the versions can hardly be seen on the plots.

But even for such a simple piece of OpenMP code, it is quite easy to introduce data races. An interesting case occurred during the development of the 4th version. While concentrating on the reduction of the number of barriers executed many times inside the iteration loop, it can easily be overseen that an additional barrier has to be introduced behind the while loop, where the value of the private variables k_priv and error_priv have to be propagated to the corresponding shared variables k and error, which are used after the parallel region (fig. 4). The modification of the variable error in the single region after the iteration loop cannot be done before all threads used it to calculate error_priv in the last iteration. An analysis with KAI’s ASSURE tool revealed that this barrier was missing initially.
3. The Thermoflow60 Finite-Element Program

Over the last 14 years a Finite Element CFD solver has been developed at the Jet Propulsion Laboratory at the Aachen University. In the early days this application was used to simulate all kind of internal and external flows. In its further development the simulation of wall heat fluxes became a new goal which was finally performed by solving the "fluid" domain and the structure domain in a coupled manner. Today the code is used to simulate heat transfer problems in rocket combustion chambers (fig. 7) [5]. Because this is a true rotational symmetric problem a 2D simulation meets the necessary accuracy. In order to determine the heat fluxes correctly, certain physical effects in the boundary layer require a very fine grid in the near wall region. This leads to grids with several 100,000 cells. These simulations can only be performed in a justifiable timeframe by using parallelization.

Because of the availability of Fujitsu-Siemens vector systems at the Aachen University throughout the nineties the program has been well adapted to the specific vector computer architecture. With the replacement of the last vector machine by a Sun Fire SMP cluster at Aachen it was necessary to parallelize the code. Fortunately OpenMP as the new de-facto standard for shared-memory parallelization is now available and mature enough to justify the investment of man power needed for parallelization andloop-level parallelization with OpenMP also is a natural replacement for vectorization, as both profit from clean data structures and long “fat” loops.

The current OpenMP version of Thermoflow60 consists of 29000 lines of Fortran with about 200 OpenMP directives, containing 69 parallel loops overall.
Expanding the parallel regions

The first OpenMP approach was quite cumbersome. Introducing parallel regions and worksharing constructs around the many loops not only was a lot of work but also very error-prone. Like other typical CFD codes written in (extended) Fortran77, all large global arrays had been put into common blocks. Many subroutines contain heavy calculations in long loops reading and modifying these global arrays with the help of many locally declared scalar temporaries, which in a loop-level parallelization have to be declared private (fig. 8). Frequently ASSURE had to be used to detect missing privatizations.

```fortran
!$omp parallel
!$omp do private(k1,k2,k3,dtel,q1,q2,q3,dtdrye,viture,ynorme,reture,
  & rhoe,tuke,eps,xe,prode1,prode2,da,pq,pr,dl,sh,sl,t,al,bs,cr,cs,
  & dl,dv,dt,dr)
!$omp& DO 21 I=1,NELM
  C
  K1 = IELM(I,1)
  K2 = IELM(I,2)
  K3 = IELM(I,3)
  C
  --- 129 lines omitted ---
```
$$q_{21} = tukl(i) \times cq1 \times cmy \times \text{dampqe} \times \text{prode1} \times \rho_{1} / \varepsilon_{1}(i)$$
$$q_{22} = tukl(i) \times cq1 \times \text{prode2}$$
$$q_{23} = tukl(i) \times cq1 \times (1.0 + 00 + sark \times xmatl) \times \varepsilon_{1}(i) / \rho_{1}(i)$$
$$w_{21} = \varepsilon_{1}(i) \times cw1e \times cmy \times \text{prode1} \times \rho_{1}(i) / \varepsilon_{1}(i)$$
$$w_{22} = \varepsilon_{1}(i) \times cw1e \times \text{cw3} \times 1.5 \times \text{prode2}$$
$$w_{23} = \varepsilon_{1}(i) \times \text{cw2} \times \varepsilon_{1}(i) / \rho_{1}(i)$$

\[ C \]
$$qtukl(i) = q_{21} + q_{22} - q_{23}$$
$$q_{\varepsilon_{1}}(i) = w_{21} + w_{22} - w_{23}$$

Fig. 8: Typical code fragment with a parallelized loop in a separate parallel region

In the following versions the parallel regions were extended. As soon as the parallel regions are extracted out of the subroutines containing the parallel loops (orphaning) the default rules for variable scoping change: Whereas in the static extent of a parallel region all these temporary scalar variables are shared by default and need to be privatized explicitly, they are private by default in the dynamic extend. Suddenly all the long lists in private clauses of the do directives vanish.

This strategy was followed until finally the whole iteration loop containing almost all of the calculation fitted into one single parallel region (corresponding to version 3 of the Jacobi example).

Avoiding the worksharing omp do construct

Revisiting the compute intensive parts, it turned out that most loops fall into two categories: loops over the number of nodes of the underlying 2D Finite Element grid on one hand (loop type 1, see below) and loops over the number of cells on the other hand (loop type 2).

If successive loops fall into the same category, there is a good chance that a barrier in between can be avoided with a nowait clause, if the loop scheduling is static, thus increasing the scalability.

Because the problematic areas of the simulation, the boundary layers, are well-known in advance, the grid has a very fine resolution in these areas from the very beginning and adaptation during the simulation process is not necessary (fig. 10).

As a consequence the loop ranges do not change over time. As in version 5 of the Jacobi example, this allows a precalculation of the loop chunks for all threads for these two loop categories (fig. 12) and the elimination of all corresponding worksharing do directives. Only the end do directives without a nowait clause have to be replaced by a barrier directive in order to insure the necessary synchronisations.
Some time consuming loop nests have a special structure which might be further exploited (loop type 3). Whereas the outer loop runs over the number of nodes, the inner loop runs over the number of cells this specific node belongs to, which may vary depending on its position in the grid. Particularly nodes on the boundaries typically belong to fewer cells than inner nodes (in many cases 4 compared to 6). This might cause an imbalance of the parallelized outer loop. As the grid does not change over time, an optimal schedule can be precalculated once. But it turned out, that in this case, the difference between an “optimal” schedule and a static work distribution does not pay off.
Fig. 12: Precomputing the loop limits
Fig. 13: Threadprivate common block containing the loop limits

Figure 12 shows the subroutine which precomputes the loop limits for all threads and stores them into a threadprivate common block (fig. 13).

```
integer MAXTHREADS
parameter ( MAXTHREADS=72 )
integer ilo_poin, ihi_poin, ilo_elm, ihi_elm
integer ilo_knot, ihi_knot
common / omp_com / ilo_poin, ihi_poin, ilo_elm, ihi_elm,
&                   ilo_knot, ihi_knot
!
C$omp threadprivate(/omp_com/)
!
DO 40 i = ilo_knot, ihi_knot
  C  do 45 j = 1,nknot(i)
  C    ii   = iknot(i,j)       ! Elementnummer
    kk   = iknel(i,j)       ! lokale Knotennummer (1-3!)
C
    ---  28 lines omitted ---
C
45   CONTINUE
40 CONTINUE
C$omp barrier
```

Fig. 14: Loop nest with precalculated optimal schedule (loop type 3)

The technique of manually load balancing of a loop nest (like those of type 3), can be generally applied to loop nests of the following structure:
```
  do i = 1, many
    do j = 1, func(i)  ! few
      call same_amount_of_work(i,j)
    end do
  end do
```

The number of executions in the loop nest body, here symbolized by the subroutine call, has just to be counted, and than evenly distributed to all threads, provided that the amount of work per inner loop iteration is constant, that the number of outer loop iterations is much higher than the number of threads and that the number of inner loop iterations is small.

**Do not put an OpenMP code into production without using Assure!**

Again and again ASSURE demonstrated its worthiness. One case was particularly remarkable (fig. 15). A loop to be parallelized contained the usage of an index array IRPKTE, which was supposed to have disjoint entries by the nature of the underlying geometry. ASSURE produced an error message, which we could not understand. Dumping out the index array to a file, and sorting it revealed: two entries out of 2000 had double counts, due to a (serial) programming error, as it turned out later.

```
c$omp do private(l,skprod)
  DO 10 I=1,NRPKTE
    L       = IRPKTE(I)
    SKPROD  = XIMP(L)*RVECTE(I,1)+YIMP(L)*RVECTE(I,2)
    XIMP(L) = XIMP(L)-SKPROD*RVECTE(I,1)
    YIMP(L) = YIMP(L)-SKPROD*RVECTE(I,2)
  10  CONTINUE
  c$omp end do
```

Fig. 15: parallel loop with an index array which is supposed to have disjoint values
Timing measurements

Timing measurements have been carried out

- on an HP V-Class model 2250 with 16 PA-RISC 8200 processors with a 240 MHz clock cycle using KAI’s Guide Preprocessor V4.0 together with HP’s Fortran90 compiler V2.5.1 under HP-UX V11,
- on a Sun Fire 6800 with 24 UltraSPARC-III/Cu processors with a 900 MHz clock cycle using Sun’s ONE Studio 7 Fortran95 compiler under Solaris 8, and
- on a Sun Fire 15K with 72 UltraSPARC-III/Cu processors with a 900 MHz clock cycle using Sun’s ONE Studio 7 Fortran95 compiler under Solaris 8.

The pre- and postprocessing phases have been neglected and 100 iterations of the compute intense part have been measured, because they clearly dominate real production runs which take many hours.

The measurements show that it is possible to write a scalable OpenMP program with “only” loop level parallelism, meaning that the worksharing is done around inner loops or loop nests in the leaves of the program tree, if the parallel region can be extended.

Of course the newer Sun Fire systems perform better than the older HP system. But the runtime can be nicely reduced by this OpenMP approach on all considered machines. A speed-up factor of over 40 on 68 CPUs on the Sun Fire 15K or equivalently an efficiency of about 60 percent and an efficiency curve which only slowly flattens out are encouraging results.

Fig. 16: Runtime of the ThermoFlow60 program on a Sun Fire 15 K, on a Sun Fire 6800 and on an HP V Class, using worksharing do directives versus precalculation of the loop ranges (omp do avoided)
Fig. 17: Efficiency of the ThermoFlow60 program on a Sun Fire 15K and on an Sun Fire 6800, using worksharing do directives versus precalculation of the loop ranges

<table>
<thead>
<tr>
<th>threads</th>
<th>Sun Open compiler</th>
<th>Sun Fire 15K</th>
<th>Guide preprocessor</th>
<th>Sun Fire 15K</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>with omp do</td>
<td>omp do avoided</td>
<td>with omp do</td>
<td>omp do avoided</td>
</tr>
<tr>
<td></td>
<td>seconds</td>
<td>speed-up</td>
<td>seconds</td>
<td>speed-up</td>
</tr>
<tr>
<td>1</td>
<td>374.176</td>
<td>0.892</td>
<td>342.744</td>
<td>0.974</td>
</tr>
<tr>
<td>2</td>
<td>193.123</td>
<td>1.729</td>
<td>176.988</td>
<td>1.886</td>
</tr>
<tr>
<td>4</td>
<td>95.474</td>
<td>3.497</td>
<td>88.299</td>
<td>3.781</td>
</tr>
<tr>
<td>6</td>
<td>62.394</td>
<td>5.350</td>
<td>58.945</td>
<td>5.663</td>
</tr>
<tr>
<td>8</td>
<td>48.936</td>
<td>7.112</td>
<td>44.307</td>
<td>7.535</td>
</tr>
<tr>
<td>10</td>
<td>38.099</td>
<td>8.762</td>
<td>36.066</td>
<td>9.256</td>
</tr>
<tr>
<td>56</td>
<td>9.613</td>
<td>34.727</td>
<td>10.141</td>
<td>32.918</td>
</tr>
<tr>
<td>70</td>
<td>8.083</td>
<td>41.299</td>
<td>9.013</td>
<td>37.041</td>
</tr>
</tbody>
</table>

Table 1: Runtime and Speed-Up of the ThermoFlow60 program on a Sun Fire 15K, with Sun’s OpenMP compiler compared to KAI’s guide preprocessor using worksharing do directives versus precalculation of the loop ranges
The measurements on both Sun Fire systems with up to 24 threads suggest that avoiding the worksharing constructs really pays off. Compiling this version with the Sun OpenMP compiler and running it on the Sun Fire 15K with only one thread almost takes the same time (342.744 seconds) as a serial program run (333.831 seconds). Thus the overhead introduced by OpenMP can be effectively reduced (see table 1).

But taking 32 and more threads the version with all the worksharing `omp do` constructs surprisingly performs better (see figure 17 and table 1). This effect also shows up when employing KAI’s guide preprocessor with the native Sun compiler underneath in comparison to the Sun OpenMP compiler. The reason is not yet understood and under current investigation.

By the way, one single experiment was made using automatic parallelization: Almost 200 loops could be parallelized by the compiler, but the speed-up was limited to 2 !

4. Conclusion

It has been demonstrated that it is possible to write a scalable OpenMP program with “only” loop level parallelism. If the parallel regions are extended and OpenMP orphaning is used to a high extent, not only the scalability of the parallelization improves, but also the default scoping seems to be more natural to a typical CFD code, saving a lot of variable privatizations.

Techniques have been demonstrated for the replacement of worksharing constructs by precalculation of the loop chunks for all threads, which may reduce the overhead of the OpenMP parallelization. This worked well for the Jacobi example and also for the Thermoflow60 Finite-Element code as long as up to 24 threads were used. Why this method is less profitable for the Thermoflow60 Finite-Element code when 32 or more threads are involved is still under investigation.

Nevertheless it can be expected that these techniques might be useful in other cases. Last but not least, KAI’s ASSURE verification tool turned out to be a very important part of the OpenMP software development cycle.

Literature

1) Center for Computing and Communication, Aachen University of Technology
   http://www.rz.rwth-aachen.de
2) Jet Propulsion Laboratory, Aachen University of Technology
   http://www.bst.rwth-aachen.de
3) KAI’s Guide OpenMP preprocessor and Assure verification tool for OpenMP programs (part of the KAP/Pro Toolset)
   http://developer.intel.com/software/products/kappro/
4) OpenMP Architecture Review Board
   http://www.openmp.org/index.cgi?samples+samples/jacobi.html