Two OpenMP Programming Patterns

Dieter an Mey
Center for Computing and Communication, Aachen University
anmey@rz.rwth-aachen.de

1 Introduction

A collection of frequently occurring OpenMP programming patterns could help an application programmer to find easy solutions for his or her parallelization task. Two proposals for a collection of such patterns are presented:
1) Reduction operations occurring in an outer loop can be efficiently parallelized by software pipelining. With this technique the Jacobi Solver for linear equations can be parallelized with only one barrier per iteration, including the error estimation.
2) A workqueuing concept has been proposed as an enhancement of the OpenMP standard [1] which might be well suited for the parallelization of a recursive function. But as long as this is not yet commonly available the implementation of a stack containing independent tasks to be executed might be helpful. This technique is applied to an adaptive integration algorithm.

2 The Jacobi Solver revisited

The OpenMP ARB's web page [2] offers an easy to understand OpenMP programming example: A small program solving the Helmholtz equation with a simple Jacobi algorithm as the hot spot. Though it might not be the most sophisticated numerical method to solve this problem, it is nevertheless quite useful to demonstrate and experiment with different OpenMP programming strategies.
Inside the iteration loop (see fig. 1) there are two loop nests which can be automatically parallelized. 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.
The program as it is presented on the above mentioned web site, shows how the OpenMP parallelization overhead can be reduced by extracting the parallel region around two parallelized loops, where today's auto-parallelizing compilers would create one parallel region for each of these loops (fig. 1).
In [3] was demonstrated how the parallel region can be further extended to contain the whole iteration loop, finally containing 3 barriers (see fig. 2).
But it seems that we cannot do with less than 3 barriers within the iteration loop. The reuse of the reduction variable `error` in each iteration generates a false dependency. Omitting one of these barriers would clearly lead to a data race: `error` has to be initialized before and protected against the summation process of the second parallel loop. A barrier is needed in order to guaranty that the initialization is finished before the first thread updates this variable. The result of the reduction process is available after the next barrier. Then it can be used in the termination condition of the iteration loop. Because the iteration loop is inside the parallel region, all threads have to take the same decision, otherwise the program will hang. So `error` cannot be initialized again for the next iteration before all threads take there decision to go on iterating. As a side effect the same barriers separate the 2 parallelized loops from each other. Copying the array `u` to `uold` has to be separated from the calculation of the new approximation of the solution `u`.

```
      error = 10.0 * tol

      k = 1

      do  ! begin of 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    ! implicit barrier

!$omp do reduction(+:error)
         do j = 2,m-1
            do i = 2,n-1
               resid = (ax*(uold(i-1,j) + uold(i+1,j))
     &                + ay*(uold(i,j-1) + uold(i,j+1))
     &                 + b * uold(i,j) - f(i,j))/b
               u(i,j) = uold(i,j) - omega * resid
               error = error + resid*resid
            end do
         enddo
!$omp end do nowait
!$omp end parallel

         k = k + 1
         error = sqrt(error)/dble(n*m)
         if (k_priv.gt.maxit .or. error_priv.le.tol) exit

      enddo ! end of iteration loop ========================================
```
Fig. 1: The Jacobi solver with one parallel region containing two parallel loops

```
!$omp parallel private(resid, k_priv,error_priv)
      k_priv = 1
      error_priv = 10.0d0 * tol
      do ! begin of iteration loop ========================================

!$omp do
         do j=1,m; do i=1,n; uold(i,j) = u(i,j); enddo; enddo
!$omp end do      ! implicit barrier --------------------------------------

!$omp single
         error = 0.0d0
!$omp end single  ! implicit barrier --------------------------------------

!$omp do reduction(+:error)
         do j = 2,m-1
            do i = 2,n-1
               resid = (ax*(uold(i-1,j) + ...) b * uold(i,j) - f(i,j))/b
               u(i,j) = uold(i,j) - omega * resid
               error = error + resid*resid
            end do
         enddo
!$omp end do      ! implicit barrier --------------------------------------

         k_priv = k_priv + 1
         error_priv = sqrt(error)/dble(n*m)
         if (k_priv.gt.maxit .or. error_priv.le.tol) exit

      enddo ! end of iteration loop ========================================
...
!$omp end parallel
```
Fig. 2: The Jacobi solver with one parallel region containing the whole iteration loop

```
!$omp parallel private(resid, k_priv,error_priv)
      k_priv = 1
      do ! begin of iteration loop ========================================

!$omp single
         error1 = 0.0d0
!$omp end single nowait

!$omp do
         do j=1,m; do i=1,n; uold(i,j) = u(i,j); enddo; enddo
!$omp end do    ! implicit barrier --------------------------------------

!$omp do reduction(+:error1)
         do j = 2,m-1
            do i = 2,n-1
               resid = (ax*(uold(i-1,j) + ...) + b * uold(i,j) - f(i,j))/b
               u(i,j) = uold(i,j) - omega * resid
               error1 = error1 + resid*resid
            end do
         enddo
!$omp end do     ! implicit barrier --------------------------------------

         k_priv = k_priv + 1
         error_priv = sqrt(error1)/dble(n*m)
         if (k_priv.gt.maxit .or. error_priv.le.tol) exit

!$omp    single
         error2 = 0.0d0
!$omp    end single nowait
!$omp    do
         do j=1,m; do i=1,n; uold(i,j) = u(i,j); enddo; enddo
!$omp    end do   ! implicit barrier --------------------------------------


!$omp    do reduction(+:error2)
         do j = 2,m-1
            do i = 2,n-1
               resid = (ax*(uold(i-1,j) + ...) + b * uold(i,j) - f(i,j))/b
               u(i,j) = uold(i,j) - omega * resid
               error2 = error2 + resid*resid
            end do
         enddo
!$omp    end do   ! implicit barrier --------------------------------------

         k_priv = k_priv + 1
         error_priv = sqrt(error2)/dble(n*m)
         if (k_priv.gt.maxit .or. error_priv.le.tol) exit

      enddo ! end of iteration loop ========================================
...
!$omp end parallel
```
Fig. 3: Unrolling and software pipelining to save another barrier

But it turns out that after manually unrolling the iteration loop twice and using two different variables for the error estimation alternately (`error1` and `error2`) software pipelining can be used (see fig. 3). By placing the usage of `error1` and the initialization of `error2` and vice versa in the same synchronization phase, one barrier per iteration can be eliminated.

So far the algorithm has not been changed, but obviously we can save the copying of the array `uold` to `u` by alternately using the values of one array, say `u1`, to calculate the new values, say `u2`, and vice versa. As a consequence we only need one barrier to protect the update of `u1` using `u2` against the update of `u2` using `u1` and the other way round. After 4-fold unrolling the iteration loop and using 4 different variables for the error estimation, the same software pipelining technique can be employed again to reduce the number of barriers per iteration to only 1 (see fig. 4).

The code is getting a bit lengthy with this technique. It can be condensed by putting the reduction variables into a vector (here: `errorh(1:3)`). Unfortunately then the reduction variable will be an array element, which is not allowed in the OpenMP reduction clause. So the reduction has to be programmed explicitly with a critical region and an explicit barrier.

```
      error1 = 0.0d0
!$omp parallel private(resid, k_priv,error_priv)
      k_priv = 1
      do ! begin of iteration loop =======================================

!$omp    single
         error2 = 0.0d0
!$omp    end single nowait
!$omp    do reduction(+:error1)
         do j = 2,m-1
            do i = 2,n-1
               resid = (ax*(u1(i-1,j) + ...) + b * u1(i,j) - f(i,j))//b
               u2(i,j) = u1(i,j) - omega * resid
               error1 = error1 + resid*resid
            end do
         enddo
!$omp    end do   ! implicit barrier -------------------------------------
         error_priv = sqrt(error1)/dble(n*m)
         if (k_priv.gt.maxit .or. error_priv.le.tol) exit
         k_priv = k_priv + 1

!$omp    single
         error3 = 0.0d0
!$omp    end single nowait
!$omp    do reduction(+:error2)
         do j = 2,m-1
            do i = 2,n-1
               resid = (ax*(u2(i-1,j) + ...) + b * u2(i,j) - f(i,j))/b
               u1(i,j) = u2(i,j) - omega * resid
               error2 = error2 + resid*resid
            end do
         enddo
!$omp    end do   ! implicit barrier -------------------------------------
         error_priv = sqrt(error2)/dble(n*m)
         if (k_priv.gt.maxit .or. error_priv.le.tol) exit
         k_priv = k_priv + 1

!$omp    single
         error4 = 0.0d0
!$omp    end single nowait
!$omp    do  reduction(+:error3)
         do j = 2,m-1
            do i = 2,n-1
               resid = (ax*(u1(i-1,j) + ...) + b * u1(i,j) - f(i,j))/b
               u2(i,j) = u1(i,j) - omega * resid
               error3 = error3 + resid*resid
            end do
         enddo
!$omp    end do   ! implicit barrier -------------------------------------
         error_priv = sqrt(error3)/dble(n*m)
         if (k_priv.gt.maxit .or. error_priv.le.tol) exit
         k_priv = k_priv + 1

!$omp    single
         error1 = 0.0d0
!$omp    end single nowait
!$omp    do reduction(+:error4)
         do j = 2,m-1
            do i = 2,n-1
               resid = (ax*(u2(i-1,j) + ...) + b * u2(i,j) - f(i,j))/b
               u1(i,j) = u2(i,j) - omega * resid
               error4 = error4 + resid*resid
            end do
         enddo
!$omp    end do   ! implicit barrier -------------------------------------
         error_priv = sqrt(error4)/dble(n*m)
         if (k_priv.gt.maxit .or. error_priv.le.tol) exit
         k_priv = k_priv + 1

      end do ! end of iteration loop =========================================
...
!$omp end parallel
```

Fig. 4: 4-fold unrolling and software pipelining to reduce the number of barriers

```
!$omp parallel private(resid,k_priv,error_priv,errorhp,khh,kh1,kh2,kh3,u0,u1)
...
      u0 = 0
      kh1 = 1
      kh2 = 2
      kh3= 3
      do ! Begin of iteration loop ========================================

!$omp    single
         errorh(kh2) = 0.0d0
!$omp    end single nowait

         errorhp = 0.0d0
!$omp    do
         do j = 2,m-1
            do i = 2,n-1
               resid = (ax*(uh(i-1,j,u0) + uh(i+1,j,u0))
     &                + ay*(uh(i,j-1,u0) + uh(i,j+1,u0))
     &                 + b * uh(i,j,u0) - f(i,j))/b
               uh(i,j,1-u0) = uh(i,j,u0) - omega * resid
               errorhp = errorhp + resid*resid
            end do
         enddo
!$omp    end do nowait
!$omp critical
         errorh(kh1) = errorh(kh1) + errorhp
!$omp end critical
!$omp    barrier    ! explicit barrier --------------------------------------

         error_priv = sqrt(errorh(kh1))/dble(n*m)
         khh = kh1
         kh1 = kh2
         kh2 = kh3
         kh3 = khh
         u0 = 1 - u0
         if (k_priv.gt.maxit .or. error_priv.le.tol) exit
         k_priv = k_priv + 1

      end do ! End of iteration loop ========================================
...
!$omp end parallel
```

Fig. 5: Trying to condense the source, complicating the compiler optimization

Also the two sweep operations between `u1` and `u2` can be combined by adding a third dimension to the u array (here: `real*8 uh(n,m,0:1)`, see fig. 5) and then alternating the value of the third dimension. But it turns out that toggling between the planes of this array might hinder the compiler optimization, because it is no longer trivial to verify that the accesses to the array are disjoint.

Table 1 shows some measurements in MFlop/s of 6 different versions of the Jacobi algorithm with a matrix size of 200x200 run on a Sun Fire 6800 with 24 UltraSPARC III Cu 900 MHz processors. The latest Sun ONE Studio 8 Fortran95 compiler was employed. The used matrices already fit in a single L2 cache, such that the memory bandwidth is not a limiting factor. On the other hand the problem is so small that it does not scale well beyond about 8 threads.

- The serial program runs at 838 MFlop/s.
- V1 is the original serial version here compiled with autoparallelization turned on. It runs at the same speed as a straight forward OpenMP version with 2 parallel regions in the iteration loop.
- V2 is the original OpenMP from [2] version with one parallel region in the iteration loop containing two parallel loops. (see fig. 1). V2 is clearly faster than V1.
- V3 contains one parallel region containing the whole iteration loop with 3 barriers (see [3] and fig 2). V3 is slightly faster than V2
- In V4 one barrier has been saved by 2-fold unrolling and software pipelining (see fig. 3). V4 is faster than V3 for more than 8 threads.
- V5 with 4-fold unrolling and software pipelining (like fig.4) plus avoiding the copy operation and toggling between two planes of a three dimensional array. This version contains only one barrier per iteration step and is by far the fastest.

- The condensed version V6 as shown in figure 5 is even slower than V1 because it could not be optimized efficiently by the compiler.

#threads	V1	V2	V3	V4	V5	V6
1	812	815	816	803	1286	333
2	1514	1534	1545	1546	2412	653
4	2624	2717	2738	2717	4335	1284
6	3314	3525	3598	3594	5717	1891
8	3655	3985	4062	4080	6529	2395
12	4078	4574	4757	5007	7443	3283
16	3861	4365	4107	4357	6552	3859

Tab. 1: Performance in MFlop/s of 6 different versions of the Jacobi algorithm

3 Parallelizing an Algorithm for Adaptive Integration

A simple algorithm for adaptive numerical integration of a real-valued function in a finite interval can easily be programmed with a recursive function in Fortran90 [4] or in C [5]. The original program is taken from the collection of examples coming with the F compiler. [4]
A main routine sets the interval boundaries, calls the integral function with the quadrature algorithm and finally prints the numerical results. The module function `f` evaluates the function to be integrated at any given point. The serial recursive function `integral` containing the quadrature algorithm is shown in figure 6.

```
recursive function integral (f, a, b, tolerance)  &
      result (integral_result)

   interface
   function f (x) result (f_result)
      real, intent (in) :: x
      real :: f_result
   end function f
   end interface
   real, intent (in) :: a, b, tolerance
   real :: integral_result
   real :: h, mid
   real :: one_trapezoid_area, two_trapezoid_area
   real :: left_area, right_area

   h = b - a
   mid = (a + b) /2
   one_trapezoid_area = h * (f(a) + f(b)) / 2.0
   two_trapezoid_area = h/2 * (f(a) + f(mid)) / 2.0 + &
                        h/2 * (f(mid) + f(b)) / 2.0
   if (abs(one_trapezoid_area - two_trapezoid_area)  &
         < 3.0 * tolerance) then
      integral_result = two_trapezoid_area
   else
      left_area = integral (f, a, mid, tolerance / 2)
      right_area = integral (f, mid, b, tolerance / 2)
      integral_result = left_area + right_area
   end if

end function integral
```

Fig. 6: The recursive function containing the quadrature algorithm.

The integral over the function `f` is approximated by two quadrature formulas: The trapezoidal rule (`one_trapezoid_area`) and the midpoint rule (`two_trapezoid_area`). The difference between these values is used to estimate the error. If the error is less than the given tolerance, the result is accepted and returned as the integral result over the given interval. Otherwise the interval is cut into halves as well as the demanded tolerance and the integral function is recursively called for both new subintervals.

```
double integral(...)
{
...
#pragma omp parallel
  {

#pragma omp taskq lastprivate(answer)
    {

#pragma omp task
  answer = integral_par(f, a, b, tolerance);
    } /* end taskq */

  } /* end parallel */
...
}
double  integral_par(
          double (*f)(double),    /* function to integrate */
          double a,         /* left interval boundary  */
          double b,         /* right interval boundary */
          double tolerance) /* error tolerance */
{
...

  h = b - a;
  mid = (a+b)/2;
  one_trapezoid_area = h * (f(a) + f(b)) / 2.0;
  two_trapezoid_area = h/2 * (f(a) + f(mid)) / 2.0 +
    h/2 * (f(mid) + f(b)) / 2.0;


  if (fabs(one_trapezoid_area - two_trapezoid_area)
          < 3.0 * tolerance){

    /* error acceptable   */
    integral_result = two_trapezoid_area;

  }else{
    /* error not acceptable */
    /* put recursiv function calls for left and right areas
       into task queue */

#pragma omp taskq
      {

#pragma omp task
        {
          left_area = integral_par(f, a, mid, tolerance / 2);
        } /* end task */

#pragma omp task
        {
          right_area = integral_par(f, mid, b, tolerance / 2);
        } /* end task */

      } /* end taskq */

      integral_result = left_area + right_area;
    }

  return integral_result;
}
```
Fig. 7: The integral module parallelized with a `taskq` construct

The KAP/Pro Toolset's guidec and guidec++ compilers and recently the Intel C++ compiler offer a non-standard OpenMP feature [1], which is well suited to parallelize this recursive algorithm very elegantly (fig. 7). An additional workqueuing concept supplements the existing OpenMP worksharing constructs. Within a parallel region a `taskq` construct contains `task` constructs or further nested `taskq` constructs. The execution of the `task` constructs will be scheduled asynchronously to the active threads at runtime.

In Fortran so far no compiler offers this `taskq` mechanism. Can this function be parallelized efficiently and somewhat elegantly with the standard OpenMP version?

The recursive function call can be replaced by a stack mechanism, used for storing the intervals and corresponding tolerances, and a loop (fig. 8). The initial interval is put on the stack. While there still are intervals on the stack, there is something to do. If the integration over the interval lying on top of the stack is successful, the stack is shrinking, if not, two new subintervals are put on the stack. Once the stack mechanism (consisting of three subroutines `new_stack`, `push`, and `pop`, and one logical function `empty_stack`) is available, the new integral function is hardly more complicated than the original version.

A first parallelization of this algorithm seems to be easy: All stack accesses have to be put into critical regions and the summation of the integral results as well.

But there still is one open problem: If somewhere in the middle of the adaptive integration process, the stack has less entries than there are threads, the spare threads will exit the loop and wait at the end of the parallel region. Actually this will already happen in the very beginning, as only the initial interval has been pushed onto the stack, such that initially only one thread has work to do. Therefore an additional counter `busy` is introduced, counting the number of threads currently actively working on an interval. Only if this counter has been set to zero, all work has been done and a thread, which encounters an empty stack will exit the loop (Fig. 9).

```
function integral (f, ah, bh, tolh)  &
      result (integral_result)

...
   type (stack_t) :: stack

   call new_stack ( stack )
   call push ( stack, ah, bh, tolh )

   integral_result = 0.0
   do
      if ( empty_stack ( stack ) ) exit
      call pop ( stack, a, b, tolerance )

      h = b - a
      mid = (a + b) /2
      one_trapezoid_area = h * (f(a) + f(b)) / 2.0
      two_trapezoid_area = h/2 * (f(a) + f(mid)) / 2.0 + &
                           h/2 * (f(mid) + f(b)) / 2.0
      if (abs(one_trapezoid_area - two_trapezoid_area)  &
            < 3.0 * tolerance) then
         integral_result = integral_result + two_trapezoid_area
      else
         call push ( stack, a, mid, tolerance / 2 )
         call push ( stack, mid, b, tolerance / 2 )
      end if

   end do

end function integral
```

Fig. 8: The integral module using a stack for the intervals to be integrated

```
function integral (f, ah, bh, tolh)  &
      result (integral_result)

...
   call new_stack ( stack )
   call push ( stack, ah, bh, tolh )
   integral_result = 0.0
   busy = 0
   ready = .false.

!$omp parallel default(none) &
!$omp shared(stack,integral_result,f,busy) &
!$omp private(a,b,tolerance,h,mid,one_trapezoid_area,&
!$omp         two_trapezoid_area,idle,ready)
   idle = .true.

   do
!$omp critical (stack)
      if ( empty_stack ( stack ) ) then
         if ( .not. idle ) then
            idle=.true.
            busy = busy - 1
         end if
         if ( busy .eq. 0 ) ready = .true.
      else
         call pop ( stack, a, b, tolerance )
         if ( idle ) then
            idle = .false.
            busy = busy + 1
         end if
      end if
!$omp end critical (stack)
      if ( idle ) then
         if ( ready ) exit
         ! call idle_loop ( 0.001 )
         cycle ! try again
      end if

      h = b - a
      mid = (a + b) /2
      one_trapezoid_area = h * (f(a) + f(b)) / 2.0
      two_trapezoid_area = h/2 * (f(a) + f(mid)) / 2.0 + &
                           h/2 * (f(mid) + f(b)) / 2.0
      if (abs(one_trapezoid_area - two_trapezoid_area)  &
            < 3.0 * tolerance) then
!$omp critical (result)
         integral_result = integral_result + two_trapezoid_area
!$omp end critical (result)
      else
!$omp critical (stack)
         call push ( stack, a, mid, tolerance / 2 )
         call push ( stack, mid, b, tolerance / 2 )
!$omp end critical (stack)
      end if
   end do
!$omp end parallel

end function integral
```

Fig. 9: The integral module parallelized with a stack.

The summation of the integral results can be further optimized, by using private variables for the partial sums, which each thread can accumulate without a critical region. Only at the very end, these partial sums are then summed up in a critical region, which is only entered once by each thread.

Finally it should be mentioned that the number of calls to the integrated function can be drastically reduced. For real life problems these function evaluations are usually by far the most time consuming parts. So far three function evaluations have to be executed during each call of the recursive function respectively during each loop iteration. If in the beginning the first function values at the initial interval boundaries are put on the stack and if then all the function values are put on the stack together with the intervals, a lot of redundant computations can be avoided. In each

loop step it is then only necessary to calculate the new function value at the midpoint of the current interval as the function values at the interval boundaries are already known from previous integration steps.
Furthermore the overhead of the usage of the stack mechanism can be easily reduced, if one of the two new intervals which is created after an unsuccessful integration is not put on the stack, but immediately handled by the same thread.

The final algorithm performs quite nicely. Of course the scalability depends heavily on the cost of the function evaluation in relation to the overhead of the stack mechanism. If this cost is varying, the algorithm automatically distributes the load among the threads. As this approach in general targets coarse-grained parallelism, its overhead will most likely play a minor role.

4 Summary

Two OpenMP programming patterns have been presented in this paper.
In chapter 2 a software pipelining technique has been presented to reduce the number of barriers in the well-known Jacobi linear equation solver. This example is particularly useful for teaching purposes, as it deals with a very error prone aspect of OpenMP program tuning: avoiding unnecessary barriers without generating data races.
In chapter 3 a stack mechanism has been developed to parallelize a recursive function for adaptive integration. This programming pattern could well be used for similar problems, like working on a linked list. It might be an input to the discussion about the necessity of the proposed `taskq` mechanism for a future version of OpenMP.
We propose to collect this kind of programming patterns on the cOMPunity web site [7].

5 References

1. Ernesto Su, et. al., “Compiler Support of the Workqueung Execution Model for Intel SMP Architectures”, EWOMP’02
2. The OpenMP Architecture Review Boards (ARB) http://www.openmp.org
3. Dieter an Mey, Thomas Haarmann, Wolfgang Koschel, “Pushing Loop-Level Parallelization to the Limit”, EWOMP’02
4. The F compiler, ftp://ftp.swcp.com/pub/walt/F/
5. Dieter an Mey, “Parallelizing an Algorithm for Adaptive Integration with OpenMP” http://www.rz.rwth-aachen.de/computing/hpc/prog/openmp/adaptive_integration.html
6. Dieter an Mey, “Parallelizing the Jacobi Algorithm” http://www.rz.rwth-aachen.de/computing/hpc/prog/openmp/jacobi.html
7. The OpenMP User Community http://www.compunity.org