
Two OpenMP Programming Patterns

EWOMP'03
Aachen
September 22 - 26, 2003

Dieter an Mey
Center for Computing and Communication
Aachen University (RWTH)

Overview

- **Introduction**
- **Jacobi**
- **Adaptive Integration**
- **Summary**

Introduction

Introduction

- **A collection of typical OpenMP programming patterns would be helpful**
 - **for teaching OpenMP and**
 - **for application programmers looking for an elegant solution**
- **Solutions realized in prototype compilers don't help the application programmer (portability!)**
- **The cOMPunity web site would be a suitable place to collect such examples:**

www.cOMPunity.org

Jacobi

www.openmp.org - Sample Program

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

```
*****
* program to solve a finite difference
* discretization of Helmholtz equation :
*  $(d^2/dx^2)u + (d^2/dy^2)u - \alpha u = f$ 
* 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 parallelism.
* All do loops are parallized with default 'static' scheduling.
*****
```

Jacobi Solver – Version 1 (1 of 2)

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 of 2)

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 (1 of 2)

Extracting the Parallel Region out of 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
        resid = (ax*(uold(i-1,j) ... )/b
        u(i,j) = uold(i,j) - omega * resid
        error = error + resid*resid
      end do; enddo
    !$omp end do
    k_priv = k_priv + 1
    error_priv = sqrt(error)/dble(n*m)
  enddo
  !$omp barrier
  !$omp single
    k = k_priv
    error = error_priv
  !$omp end single nowait
!$omp end parallel
```

FORK

BARRIER

BARRIER

BARRIER

BARRIER

JOIN

Looking closer at a Reduction (1 of 5)

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

Standard case: OK

Looking closer at a Reduction (2 of 5)

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

Standard case: OK

Reduction variable has to be initialized inside a larger parallel region:

Assure: Error

Thread Checker: Error

Wrong results: Sun, HP, PGI, Intel, SGI

So far no wrong results: Guide, IBM

Looking closer at a Reduction (3 of 5)

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

Standard case: OK

Reduction variable has to be initialized

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

Assure: Error

Thread Checker: No Error (bug!)

Sun compiler prints OMP warning
(if activate)

Wrong results: Sun, HP, PGI, Intel, SG

So far no wrong results: Guide, IBM

Looking closer at a Reduction (4 of 5)

```
error = 0.0
```

```
!$omp parallel
```

```
!$omp parallel
```

```
..
```

```
!$omp parallel
```

```
!$omp
```

```
!$omp parallel
```

```
!$omp
```

```
!$omp
```

```
!$omp single
```

```
!$omp
```

```
error = 0.0
```

```
!$omp
```

```
!$omp end single
```

```
!$omp
```

```
!$omp do reduction(+:error)
```

```
!$omp
```

```
do ..
```

```
!$omp
```

```
error = error + ..
```

```
!$omp
```

```
end do
```

```
!$omp
```

```
!$omp end do
```

```
!$omp
```

```
.. func ( error ) ..
```

```
!$omp
```

```
..
```

```
!$omp
```

```
!$omp end parallel
```

Standard case: OK

Reduction variable has to be initialized

Reduction variable is initialized

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

Assure: OK
Thread Checker: OK

Always correct results

Looking closer at a Reduction (5 of 5)

<pre>error = 0.0 !\$omp parallel</pre>		Standard case: OK
<pre>!\$omp parallel ..</pre>		Reduction variable has to be initialized
<pre>!\$omp parallel !\$omp barrier</pre>		Reduction variable is initialized
<pre>!\$omp parallel !\$omp barrier !\$omp barrier</pre>		Reduction variable is initialized
<pre>!\$omp parallel !\$omp barrier !\$omp barrier !\$omp barrier</pre>		Reduction variable is initialized
<pre>!\$omp parallel !\$omp barrier !\$omp barrier !\$omp barrier !\$omp barrier</pre>	<pre>error = 0.0 !\$omp barrier !\$omp do reduction(+:error) do .. error = error + A end do !\$omp end do .. func (error) ..</pre>	Redundantly with an explicit barrier
<pre>!\$omp parallel !\$omp barrier !\$omp barrier !\$omp barrier !\$omp barrier !\$omp barrier</pre>	<pre>do .. error = error + A end do</pre>	Assure: Error Thread Checker: Error
<pre>!\$omp parallel !\$omp barrier !\$omp barrier !\$omp barrier !\$omp barrier !\$omp barrier !\$omp barrier</pre>	<pre>.. func (error) ..</pre>	(yes, there is a data race, but..)
<pre>!\$omp parallel !\$omp barrier !\$omp barrier !\$omp barrier !\$omp barrier !\$omp barrier !\$omp barrier !\$omp barrier</pre>	<pre>!\$omp end parallel</pre>	Correct results

Jacobi Solver – Version 3 (2 of 2)

Extracting the Parallel Region out of the Iteration Loop

```
!$omp parallel private(..)
  ..
    !$omp do
      .. uold.. = u ..
BARRIER !$omp end do
    !$omp single
BARRIER error = 0.0
    !$omp end single
BARRIER !$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
BARRIER error = 0.0
    !$omp end single
BARRIER !$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 (1 of 2)

Software Pipelining

```
!$omp parallel private(..)
  ..
    !$omp do
      .. uold.. = u ..
BARRIER !$omp end do
    !$omp single
BARRIER error1 = 0.0
    !$omp end single
BARRIER !$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
BARRIER error2 = 0.0
    !$omp end single
BARRIER !$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
```

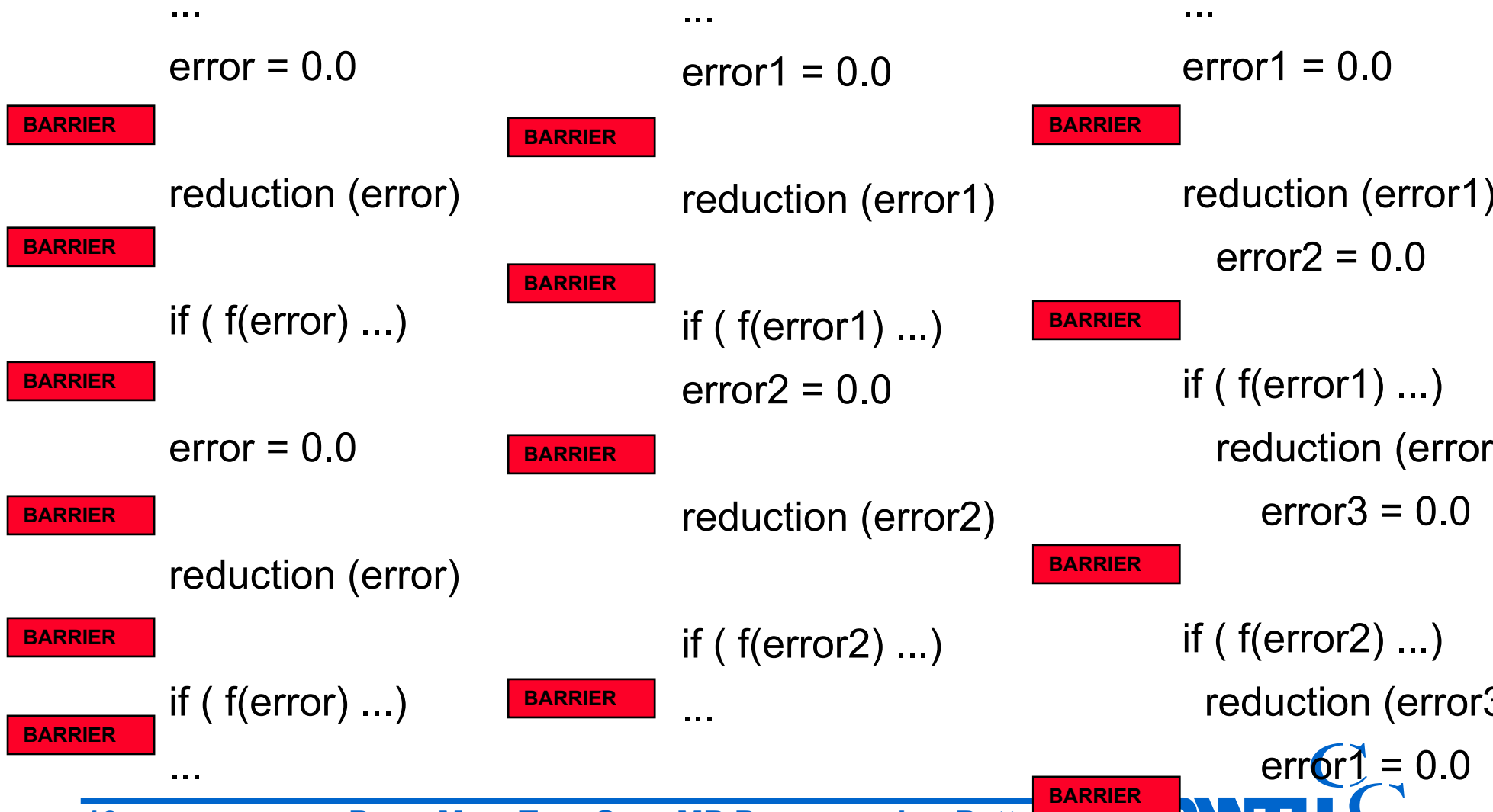
Jacobi Solver – Version 4 (2 of 2)

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 (1 of 2)

Software Pipelining



Jacobi Solver – Version 5 (1 of 2)

Software Pipelining

```
!$omp parallel private(..)
```

```
..
```

```
!$omp single
```

```
error1 = 0.0
```

```
BARRIER
```

```
!$omp end single
```

```
!$omp do reduction(+:error)
```

```
do j,i..; u1(i,j)=u2(i,j)..;error1=error1 + ..; end do
```

```
BARRIER
```

```
!$omp end do
```

```
.. if ( f(error1) ) exit
```

```
!$omp single
```

```
error2 = 0.0
```

```
BARRIER
```

```
!$omp end single
```

```
!$omp do reduction(+:error)
```

```
do j,i..; u2(i,j)=u1(i,j)..;error2=error2 + ..; end do
```

```
BARRIER
```

```
!$omp end do
```

```
.. if ( f(error2) ) exit
```

```
..
```

```
!$omp end parallel
```

Jacobi Solver – Version 5 (2 of 2)

Software Pipelining

```
!$omp parallel private(..)
```

```
..
```

```
!$omp single
```

```
error2 = 0.0
```

```
!$omp end single nowait
```

```
!$omp do reduction(+:error)
```

```
do j,i..; u1(i,j)=u2(i,j)..;error1=error1 + ..; end do
```

BARRIER

```
!$omp end do
```

```
.. if ( f(error1) ) exit
```

```
!$omp single
```

```
error3 = 0.0
```

```
!$omp end single nowait
```

```
!$omp do reduction(+:error)
```

```
do j,i..; u2(i,j)=u1(i,j)..;error2=error2 + ..; end do
```

BARRIER

```
!$omp end do
```

```
.. if ( f(error2) ) exit
```

```
..
```

```
..
```

Jacobi Solver – Version 6

Trying to condense the source code

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

The compiler might no longer recognize, that there is no data dependency between $uh(\dots, 1-u0)$ and $uh(\dots, u0)$

Jacobi Solver - Comparison

#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

Mflop/s, Matrix size 200x200
Sun Fire 6800, US-III Cu 900 MHz, f90 V7.1, Solaris 9

Jacobi Solver - Comparison

#threads	V1	V2	V3	V4	V5	V6
1	1997	1337	1330	1329	1849	999
2	3786	2542	2570	2576	3560	1958
4	6825	4554	4605	4636	6438	3668
6	9474	6246	6270	6438	9018	5327
8	11079	7677	7482	7949	10958	6745
12	12181	9381	9180	9199	13892	8803
16	8660	8653	6245	8531	13814	9439

Mflop/s, Matrix size 200x200
IBM Regatta, Power4 1700 MHz, Fortran XL V8.1.1, AIX 5.1

Adaptive Integration

Adaptive Integration

Serial Version – Recursive Function

```
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
  ..
  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
```

<ftp://ftp.swcp.com/pub/walt/F/>

Adaptive Integration

Parallelization with Taskqueue – Guide / Intel C/C++ only

```
double integral
{
    ...
    #pragma omp para
    {
        #pragma omp task
        {
            #pragma omp taskwait
            answer =
        } /* end task
    } /* end para
    ...
}
```

```
double integral_par( double (*f)(double), double a, double b, double 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 recursive function calls for left and right areas into taskqueue */
        #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;
}
```

Adaptive Integration

Stack Mechanism - Serial Version

```
function integral (f, ah, bh, tolerance) result (result)
...
type (stack_t) :: stack
call new_stack ( stack )
call push ( stack, ah, bh, tolerance )
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
```

Adaptive Integration

Stack Mechanism - OpenMP Version (1 of 2)

```
call new_stack ( stack )
call push ( stack, ah, bh, tolh )
integral_result = 0.0
!$omp parallel private(a,b,tolerance,h,mid,one_trapezoid_area, two_trapezoid_area, ready)
ready = .false.
do
!$omp critical (stack)
    if ( empty_stack ( stack ) ) then; ready = .true.
    else; call pop ( stack, a, b, tolerance )
    end if
!$omp end critical (stack)
    if ( ready ) exit
    h = b - a
    mid = (a + b) / 2
    one_trapezoid_area = h * (f(a) + f(b)) / 2.0
    two_trapezoid_area = h/2 * (f(a) + f(mid)) / 2.0 + h/2 * (f(mid) + f(b)) / 2.0
    if (abs(one_trapezoid_area - two_trapezoid_area) < 3.0 * tolerance) then
!$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
```

The naive approach:
„If stack empty, exit“
does not work, because all
but one threads will stop
immediately

Adaptive Integration

Stack Mechanism - OpenMP Version (2 of 2)

```
..
  b:
!$omp
  two_trapezoid_area, rate, ready,
  ready = .false.
  idle = .true.
  do
!$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
!$omp end critical (stack)
    if ( idle ) then
      if ( ready ) exit
      cycle ! try again (delay?)
    end if
    .....
!$omp end parallel end function integral
```

Stop if stack is empty and
no thread is busy any more.

Further Improvements ...

- Eliminate critical region for the summation in inner loop.
- Reduce number of function calls by storing the function values together with the interval limits in the stack.
- Reduce overhead of the stack mechanism by immediately handling of one of the two new intervals after an unsuccessful integration by the same thread.

- Scalability depends heavily on the cost of the function evaluation in relation to the overhead of the stack mechanism.
- Load balancing is included because 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.

Summary

Summary

Two OpenMP programming patterns have been presented

- Jacobi
 - reduction in an outer loop
 - reduce the number of barriers
 - circling through a set of reduction variables
 - software pipelining technique
 - prevent data races
- Adaptive Integration
 - stack mechanism for parallelizing a recursive function
 - further tuning measures
- Collection of programming patterns on the cOMPunity web site proposed.