`omp critical`

The code inside a CRITICAL region is executed by only one thread at a time. The order is not specified. This means that if a thread is currently executing inside a CRITICAL region and another thread reaches that CRITICAL region and attempts to execute it, it will block until the first thread exits that CRITICAL region.

```
#pragma omp critical
  x = x + 1;

!$OMP CRITICAL
      X = X + 1
!$OMP END CRITICAL
```

(Very inefficiant) example using `critical` instead of `reduction`:

```
#pragma omp parallel for
  for (int i=0; i < n; i++)
    {
    double product = a[i] * b[i];
#pragma omp critical
    result=result + product;
    }
```

`omp atomic`

- If a code line is defined as atomic then this tells the compiler that the variable update in this line is to be viewed as an entity that may only be executed completely.
- This will prevent other threads to interfere with the operations of this line. It is like a mini-`critical`. There are a lot of limitations for for `atomic` clause (not for functions, basically only for expressions that can be compiled to a single machine instruction) so we will not discuss details here.

```
#pragma omp parallel for
  for (int i=0; i < n; i++)
    {
    double product = a[i] * b[i];
#pragma omp atomic
    result=result + product;
    }
```

Examples

Let us regard two simple OpenMP examples in C and FORTRAN. Fortran 90: use omp_lib

```
!FORTRAN
!$omp parallel default(none)
!$omp shared(n,x,y) private(i)
!$omp do
      do i=1,n
         x(i)=x(i)+y(i)
      end do
!$omp end do
!$omp end parallel
```

Example Matrix times vector:

```
#pragma omp parallel for default(none) private(i,j,sum) shared(m,n,a,b,c)
for(i=0;i<m;i++)
  {
  sum=0.0;
  for(j=0;j<n;j++)
    {
    sum+=b[i][j]*c[j];
    }
  a[i]=sum;
  }
```

Here the outer loop is split among the processors

Some more functions in `<omp.h>`.

```
int ID=omp_get_thread_num();
omp_set_num_threads(4);
omp_in_parallel(); // true if in parallel region
omp_get_max_threads();
```

Usage e.g.:

```
#include <omp.h>

void main()
 {
#pragma omp parallel
   {
     int id=omp_get_thread_num();
     printf(``Hello %d'',id);
```

```
    printf(‘‘world %d’’,id);
  }
}
```

Mğlicher Ausgabe:
Hello(1)Hello(0)world(1)world(0)
Hello(2)Hello(3)world(3)world(2)

Static and Dynamic Scheduling

Static scheduling (the default): Upon encountering the directive the loop is divided into chunks of the same size

```
#pragma omp for schedule(static)
for(int n=0;n<10;n++)
   {
   printf("%d", n);
   }
```

Dynamic scheduling: No predictable order for the loop. Every thread askes the scheduler for an index then computes and asks for the next index. This is useful when the iterations may take different time to execute. The overhead is much larger than in static scheduling!

```
#pragma omp for schedule(dynamic)
for(int n=0;n<10;n++)
   {
   printf("%d", n);
   }
```

Dynamic scheduling using a chunk size: Every thread askes the scheduler for a chunk of the specified size then computes the iterations and asks for the next chunk. This is useful to reduce the number of calls to the scheduler.

```
int chunk=2;
#pragma omp for schedule(dynamic, chunk)
for(int n=0;n<10;n++)
   {
   printf("%d", n);
   }
```

Sometimes it is necessary to specify an order for a part of the parallel region:

```
#pragma omp for schedule(dynamic, 2)
for(int n=0;n<10;n++)
   {
   printf("%d", n);
#pragma omp ordered
   printf("%d", n);
   }
```

`omp parallel numthreads`

The number of threads can also be specified in the `parallel` directive:

```
#omp parallel numthreads(2)
```

`omp parallel for firstprivate lastprivate`

Private Variables are uninitialized upon entry and after leaving a parallel region. Sometimes you may want to initialize the private variable, e.g. with a value from the master. This can be done using `firstprivate`. The value is then used as initialization value for all threads. Example:

```
int i=1,j;
prinff("i=%d,j=%d\n",i,j);
#pragma omp parallel firstprivate(i) lastprivate(j)
for(j=i;j<10;j++)
  {
  printf("i=%d,j=%d\n",i,j);
  }
printf("i=%d,j=%d\n",i,j);
```

Note that the firstprivate-value is set only at initialization, e.g. when the parallel region is entered. Using `lastprivate` the thread that handles the last index (here $j == 9$) or last `section` will copy the value of the lastprivate-variable to the shared variable with the same name. Arrays can also be firstprivate and lastprivate. In this case all elements of the array are copied (very untypical behavior for C/C++);

_OPENMP

The Compiler macro `_OPENMP` is defined to the date of the OpenMP version. So calls to functions in `<omp.h>` can be surrounded by

```
#ifdef _OPENMP
  tid=omp_get_thread_num();
  max=omp_get_max_threads();
  omp_set_num_threads(max);
  truefalse=omp_in_parallel(max);
#endif
```

omp parallel numthreads

```
#pragma omp parallel numthreads(2)
  {
  ...
  }
```

Pointers

Pointers to shared variables are okay but pointers to private variables may not be exchanged in between threads. The result is undefined.

OpenMP Sections

One can also parallelize totally unrelated parts of a program.

```
#pragma omp parallel sections
{
  {
  Part1();
  }
  #pragma omp section
  {
  Part2();
  Part3();
  }
  #pragma omp section
  {
  Part4();
  }
}
```

`Part2` and `Part3` run in sequence but parallel to `Part1` and `Part4`. If there are more sections than threads remaining sections will be executed later.

Conditional parallelization: `pragma omp parallel if`

Since OpenMP is efficient only from a certain problem size on an `if`-clause can be added to the `parallel` directive.

```
#pragma omp parallel for if(x.size() > 1000)
//...
```

Nested Parallel Loops

The standard behavior is that upon entering a nested parallel region a new team consisting of only one thread is constructed. Thus the total number of threads will not change.

By calling

```
omp_set_nested(1);
```

one can enable nested parallel regions/loops.

Upon encouting a second `#pragma omp parallel for` each thread will spawn N threads resulting in $N * N$ threads.

Make sure that this is what you want; you may want to call

```
omp_set_thread_num(2);
```

so that the number of threads does not explode. Instead of having nested loops one may want to remove the nesting.

Example: OpenMP Parallel LU-Decomposition

```
for(k=0; k<size-1; k++)
  {
#pragma omp parallel for default(none) shared(M,L,size,k) private(i,j) schedule(sta
  for(i=k+1; i<size; i++)
     {
     l[i][k] = a[i][k] / a[k][k];
     for (j=k+1; j<size; j++)
        a[i][j] = a[i][j] - l[i][k]*a[k][j];
     }
  }
```

parallel qsort using nested parallelism?

Global variables and COMMON blocks

The `threadprivate` directive is used to make global variables or COMMON blocks local to a thread. It is global within the thread (and persistent through several parallel regions) but not shared be the threads.

```
int a;
#pragma omp threadprivate(a)

int main()
{
//...
}
```

omp parallel copyin

Can be used to assign a value from the master thread to all threadprivate variables.

```
   a=10;  // on the master
#pragma omp parallel copyin(a)
 {
 // each private variable a is initialized by the value from the master
 }
```

The master thread variable is used as the copy source and the team threads are initialized with its value upon entry into the parallel construct.

Also see: #pragma omp parallel copyprivate (can be used with single)

Locking

Variables can explicitely be locked for all threads:

```
int a=1;
omp_lock_t l;
omp_init_lock(&l); // tells the OpenMP that you want to
                   // use locking on this variable
omp_set_lock(&l);    // locks; waits if already locked
omp_unset_lock(&l);  // unlocks; only the same thread can unlock
omp_test_lock(&l);   // tests if variable is locked;
                   // if not it locks and returns false;
                   // if yes it returns true and does not wait
omp_destroy_lock(&l); // variable is released from locking management
```

Caveat: It is easy to produce deadlocks! There is also nested locking! Here, sons in the calling tree are allowed to lock (and therefore modify) the variables that have been locked by the father (but not by others). Therefore a thread can lock the same variable several times.

After the first day on OpenMP (and Exercise 3) you may have thought that OpenMP is very simple and waterproof. I showed you some the things just to convince you that OpenMP can also be very complicated. This is important before we start with MPI - which definetely is complicated.

Deadlocks: Problem of the Philosphers

To ilucidate the problem of deadlocks: The problem of the philosophers was first formulated by Dijkstra. Five philosophers are sitting around a table each with a bowl of (parboiled) rice. There is no cutlery but just chop sticks. There are only five chop sticks on the table - one between each philosopher. Regard the following algorithm:

- Take the chop stick left of you.
- take the chop stick right of you.
- Eat.
- Put down the chop stick on the left side.
- Put down the chop stick on the right side.

Executing this algorithm in parallel will lead to a deadlock.

In OpenMP this can happen if inside a critical region a function is called which contains another critical region. In this case the critical region of the called function will wait for the first critical region to terminate - which will never happen. Giving different names (in brackets) to the critical regions helps in this situation. Critical regions of the a different name are allowed to run in parallel.

It is solely the programmers responsibility to deal with deadlocks!

Debugging using the GNU Debugger (gdb)

Compile with debugging information `-g`. Usually `-O` (optimize) is omitted with `-g`. The GNU compiler allows the use of both.

On the cluster the module of `gdb` must be loaded first

```
module load gdb/7.6
```

```
> gcc -g -O matrixmult_seq_simple_2013.c -o matrixmult_seq_simple_2013
```

Start debugger, `r` or `run` will run the program (command line arguments can be given to `r`).

```
> gdb matrixmult_seq_simple_2013
GNU gdb (GDB; openSUSE 11.1) 6.8.50.20081120-cvs
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-suse-linux".
For bug reporting instructions, please see:
<http://bugs.opensuse.org/>...
```

```
(gdb) r
Program exited with code 0377.
```

Setting Breakpoints

The command `b` or `break` will set a break point in the program. Arguments can be function names or line numbers

```
(gdb) b main
Breakpoint 1 at 0x4005dc: file matrixmult_seq_simple_2013.c, line 25.
(gdb) b 44
Breakpoint 2 at 0x4006de: file matrixmult_seq_simple_2013.c, line 44.
```

Running the Program inside the Debugger

Now `r` or `run` will run the program up to the break point

```
(gdb) r
Starting program: /home/rheinbach/freiberg/lehre/high_performance_computing_WS2013/

Breakpoint 1, main () at matrixmult_seq_simple_2013.c:25
25      {
```

Continuing

Now `c` or `continue` will continue to the next break point

```
(gdb) c
Continuing.
Breakpoint 2, main () at matrixmult_seq_simple_2013.c:44
44        for(i=0;i<N;i++)
```

List the Source

The command `l` or `list` will show the surrounding lines

```
(gdb) l
39            b[i*N+j]= 1.0/(i+j+1);
40            c[i*N+j]= 0;
41            }
42          }
43
44        for(i=0;i<N;i++)
45          {
46            for(j=0;j<N;j++)
47              {
48                c[i*N+j]=0;
```

Printing Variables

Using the command `p` (print) the content of variables can be shown

```
(gdb) p i
$1 = 2000
(gdb) p N
$2 = 2000
```

Note that with `-O` something like this might happen

```
(gdb) p i
$1 = <value optimized out>
```

Printing Arrays

If `a` is an array (=pointer) printing `a` will only print the address,

```
(gdb) p a
$7 = (double *) 0x7ffff5c02010
```

`p *a` or `p a[0]` will only print the first entry of `a`.

```
(gdb) p *a
$8 = 1
(gdb) p a[0]
$9 = 1
```

To print 20 entries of `a` use

```
(gdb) p *a@20
$10 = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20}
```

Next

Using `n` or `next` you can run the program line by line

```
(gdb) n
46              for(j=0;j<N;j++)
(gdb) n
48                  c[i*N+j]=0;
(gdb) n
49                  for(k=0;k<N;k++)
(gdb) n
51                      c[i*N+j]+=a[i*N+k]*b[k*N+j];
```

Step

The command `d` or `step` will also run the program line by line – but it will *step into* function calls.

Attention: You may find yourself debugging `printf` which you most certainly do not want.

```
(gdb) s
51                    c[i*N+j]+=a[i*N+k]*b[k*N+j];
(gdb) s
49                for(k=0;k<N;k++)
(gdb) s
51                    c[i*N+j]+=a[i*N+k]*b[k*N+j];
(gdb) s
49                for(k=0;k<N;k++)
```

More Sophisticated Breakpoints

```
break 100 if i<10
```

breaks if the condition is met.

```
tbreak 200
```

breaks only once (temporary).

```
info break
```

Shows all break points. Breakpoints can be deleted using `delete`

```
(gdb) info break
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x000000000040065a in main at matrixmult_seq_simple
        breakpoint already hit 1 time
2       hw watchpoint  keep y                      k
        breakpoint already hit 2 times
3       breakpoint     keep y   0x000000000040065a in main at matrixmult_seq_simple
4       breakpoint     keep y   0x000000000040065a in main at matrixmult_seq_simple
(gdb) delete 2
```

Watching Variables

```
watch i
```

Watchpoint: stops when a value is written to `i`.

```
rwatch i
```

Read Watchpoint: stops when a value is read to `i`.

```
awatch i
```

All Watchpoint: stops on read or write to `i`.

Example for Watching Variables

```
(gdb) watch k
Hardware watchpoint 2: k
(gdb) c
Continuing.
Hardware watchpoint 2: k

Old value = 0
New value = 1
0x0000000000400830 in main () at matrixmult_seq_simple_2013.c:49
49                  for(k=0;k<N;k++)
(gdb) c
Continuing.
Hardware watchpoint 2: k

Old value = 1
New value = 2
0x0000000000400830 in main () at matrixmult_seq_simple_2013.c:49
49                  for(k=0;k<N;k++)
```

Input and Output Redirection in Run

If your program reads input from stdin you can redirect the input using “¡”. You can pipe the output to a file using “¿”.

```
r < inputfile >outputfile
```

There is always Help

```
(gdb) help
List of classes of commands:

aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
internals -- Maintenance commands
obscure -- Obscure features
running -- Running the program
stack -- Examining the stack
status -- Status inquiries
support -- Support facilities
tracepoints -- Tracing of program execution without stopping the program
user-defined -- User-defined commands
```

Basics of Parallel Debugging using the GNU Debugger (gdb)

Consider the follwing program

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

// compile: gcc -O -fopenmp matrixmult_2013.c

void multiply(const int N, const double *a, const double *b, double *c)
{
  int i,j,k;

#pragma omp parallel for private(i,j,k)
  for(i=0;i<N;i++)
    {
      for(j=0;j<N;j++)
        {
          c[i*N+j]=0;
          for(k=0;k<N;k++)
            {
              c[i*N+j]+=a[i*N+k]*b[k*N+j];
            }
        }
    }
}

void print(const double *a, int N)
{
  {
    int i,j;
    for(i=0;i<N;i++)
      {
        for(j=0;j<N;j++)
          {
            printf(" %f",a[i*N+j]);
            //std::cout<<" "<<a[i*N+j];
          }
        printf("\n");
      }
    printf("\n");
  }
}

int main()
{
  const int N=2000;
  //const int N=3;
  int i,j,k;

  double *a=(double*)malloc(sizeof(double)*N*N);
  double *b=(double*)malloc(sizeof(double)*N*N);
  double *c=(double*)malloc(sizeof(double)*N*N);

#pragma omp parallel
   {
   int numthreads,num;
   numthreads=omp_get_num_threads();
   num=omp_get_thread_num();
   printf("Thread %d of %d\n",num,numthreads); // <----- Breakpoint 1
   }
```

```
#pragma omp parallel for private(i,j)
  for (i=0; i<N; i++)
    {
    for (j=0; j<N; j++)
      {
      a[i*N+j]= i+j+1.0;
      b[i*N+j]= 1.0/(i+j+1);
      c[i*N+j]= 0;
      }
    }

  for(i=0;i<1;i++)
    {
    multiply(N,a,b,c);
    }

#pragma omp master
  {
    if(N<20)  // <----- Breakpoint 2
      {
        print(a,N);
        print(b,N);
        print(c,N);
      }
    else
      {
        printf("N too big. I do not print.\n");
      }
  }
}
```

Breakpoints

Breakpoints can be set as in sequential programs – they are valid for ALL threads!

```
(gdb) b 57
(gdb) b 78
```

The debugger will inform you when threads are being created

```
(gdb) r
Starting program: /home/rheinba/a.out
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
[New Thread 0x40200940 (LWP 24403)]
[New Thread 0x40401940 (LWP 24404)]
[New Thread 0x40602940 (LWP 24405)]

Breakpoint 1, main.omp_fn.1 (.omp_data_i=0x0) at matrixmult_2013.c:57
57          printf("Thread %d of %d\n",num,numthreads);
```

Next

Now `next` will continue to the next line and switch to the next thread

```
(gdb) n
[Switching to Thread 0x40602940 (LWP 24405)]

Breakpoint 1, main.omp_fn.1 (.omp_data_i=0x0) at matrixmult_2013.c:57
57          printf("Thread %d of %d\n",num,numthreads);
(gdb) n
Thread 0 of 4
[Switching to Thread 0x40200940 (LWP 24403)]

Breakpoint 1, main.omp_fn.1 (.omp_data_i=0x0) at matrixmult_2013.c:57
57          printf("Thread %d of %d\n",num,numthreads);
(gdb) n
Thread 3 of 4
[Switching to Thread 0x40401940 (LWP 24404)]

Breakpoint 1, main.omp_fn.1 (.omp_data_i=0x0) at matrixmult_2013.c:57
57          printf("Thread %d of %d\n",num,numthreads);
(gdb) n
```

```
Thread 1 of 4
Thread 2 of 4
52      #pragma omp parallel
```

As usual `continue` will continue to the next breakpoint

```
(gdb) c
Continuing.

Breakpoint 4, main () at matrixmult_2013.c:78
78          if(N<20)
```

Information about the Existing Threads

You can get information about the existing threads. Here we see that only the master is doing something all other threads are waiting in `libgomp.so`.

```
(gdb) info threads
  Id   Target Id         Frame
  4    Thread 0x40602940 (LWP 24841) 0x00002aaaaacd14fe in ?? () from /usr/lib64/li
  3    Thread 0x40401940 (LWP 24840) 0x00002aaaaacd14fe in ?? () from /usr/lib64/li
  2    Thread 0x40200940 (LWP 24839) 0x00002aaaaacd14f8 in ?? () from /usr/lib64/li
* 1    Thread 0x2aaaab652b70 (LWP 24838) main () at matrixmult_2013.c:78
```

You can set breakpoints only for specific threads – but the thread must already exist.

```
(gdb) b 58 thread 4
Breakpoint 7 at 0x400b46: file matrixmult_2013.c, line 58.
```

`info break` helps you organize the breakpoints.

```
(gdb) info break
Num     Type           Disp Enb Address            What
7       breakpoint     keep y   0x0000000000400b46 in main.omp_fn.2 at matrixmult_2
        stop only in thread 4
```

Profiling using gprof

Profiling tries to determine where time is spent inside a program. First profile then decide where to optimize/parallelize

Compile with profiling information `-pg` and run.

```
g++ -pg -O mytest.c;
./a.out
```

A file `gmon.out` will be created by your program.

You can now view the profile (flat profile)

```
gprof ./a.out

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
100.33     58.08    58.08        1    58.08    58.08  print
  0.19     58.19     0.11        3     0.04    19.40  multiply
```

You can view an annotated source (needs `-g`) by `gprof -A ./a.out`.