# Lecture 10: Introduction to OpenMP (Part 2)

# Performance Issues I

- C/C++ stores matrices in row-major fashion.
- Loop interchanges may increase cache locality

```
{
  …
        #pragma omp parallel for
        for(i=0;i< N; i++)
        {
              for(j=0;j< M; j++)
              {
                  A[i][j] =B[i][j] + C[i][j];
              }
         }
}
```

- Parallelize outer-most loop

# Performance Issues II

- Move synchronization points outwards. The inner loop is parallelized.
- In each iteration step of the outer loop, a parallel region is created. This causes parallelization overhead.

```
{
  …
        for(i=0;i< N; i++)
        {
               #pragma omp parallel for
               for(j=0;j< M; j++)
               {
                  A[i][j] =B[i][j] + C[i][j];
               }
        }
}
```

# Performance Issues III

- Avoid parallel overhead at low iteration counts

```
{
   …

    #pragma omp parallel for if(M > 800)
     for(j=0;j< M; j++)
      {
             aa[j] =alpha*bb[j] + cc[j];
      }
}
```

# C++: Random Access Iterators Loops

- Parallelization of random access iterator loops is supported

```
void iterator_example(){
   std::vector vec(23);
   std::vector::iterator  it;

   #pragma omp parallel for default(none) shared(vec)
   for(it=vec.begin();  it< vec.end(); it++)
   {
         // do work with it //
    }
}
```

# Conditional Compilation

- Keep sequential and parallel programs as a single source code

```
#if def _OPENMP
#include "omp.h"
#endif

Main()
{
 #ifdef _OPENMP
   omp_set_num_threads(3);
#endif
           for(i=0;i< N; i++)
           {
                   #pragma omp parallel for
                   for(j=0;j< M; j++)
                  {
                      A[i][j] =B[i][j] + C[i][j];
                  }
            }
}
```

# Be Careful with Data Dependences

- Whenever a statement in a program reads or writes a memory location and another statement reads or writes the same memory location, and at least one of the two statements writes the location, then there is a data dependence on that memory location between the two statements. The loop may not be executed in parallel.

```
for(i=1;i< N; i++)
{
    a[i] = a[i] + a[i-1];
}
```

a[i] is written in loop iteration i and read in loop iteration i+1. This loop can not be executed in parallel. The results may not be correct.

# Classification of Data Dependences

- A data dependence is called loop-carried if the two statements involved in the dependence occur in different iterations of the loop.
- Let the statement executed earlier in the sequential execution be loop S1 and let the later statement be S2.
  - Flow dependence: the memory location is written in S1 and read in S2. S1 executes before S2 to produce the value that is consumed in S2.
  - Anti-dependence: The memory location is read in S1 and written in S2.
  - Output dependence: The memory location is written in both statements S1 and S2.

- Anti-dependence

```
for(i=0;i< N-1; i++)
{
   x = b[i] + c[i];
   a[i] = a[i+1] + x;
}
```

- Parallel version with dependence removed

```
#pragma omp parallel for shared (a, a2)
for(i=0; i < N-1; i++)
    a2[i] = a[i+1];
#pragma omp parallel for shared (a, a2) lastprivate(x)
for(i=0;i< N-1; i++)
{
   x = b[i] + c[i];
   a[i] = a2[i] + x;
}
```

```
for(i=1;i< m; i++)
    for(j=0;j<n;j++)
{
    a[i][j] = 2.0*a[i-1][j];
}
```

```
for(i=1;i< m; i++)
 #pragma omp parallel for
    for(j=0;j<n;j++)
{
    a[i][j] = 2.0*a[i-1][j];
}
```

Poor performance, it requires m-1 fork/join steps.

```
#pragma omp parallel for private (i)
for(j=0;j< n; j++)
  for(i=1;i<m;i++)
{
    a[i][j] = 2.0*a[i-1][j];
}
```

Invert loop to yield better performance.

- Flow dependence is in general difficult to be removed.

```
X = 0.0;
for(i=0;i< N; i++)
{
   X = X + a[i];
}
```

```
X = 0.0;
#pragma omp parallel for  reduction(+:x)
for(i=0;i< N-1; i++)
{
   x = x + a[i];
}
```

- Elimination of induction variables.

```
idx = N/2+1; isum = 0; pow2 = 1;
for(i=0;i< N/2; i++)
{
   a[i] = a[i] + a[idx];
   b[i] = isum;
   c[i] = pow2;
   idx++; isum += i; pow2 *=2;
}
```

- Parallel version

```
#pragma omp parallel for shared (a,b)
for(i=0;i< N/2; i++)
{
   a[i] = a[i] + a[i+N/2];
   b[i] = i*(i-1)/2;
   c[i] = pow(2,i);
}
```

- Remove flow dependence using loop skewing

```
for(i=1;i< N; i++)
{
   b[i] = b[i] + a[i-1];
   a[i] = a[i]+c[i];
}
```

- Parallel version

```
b[1]=b[1]+a[0];
#pragma omp parallel for shared (a,b,c)
for(i=1;i< N-1; i++)
{
   a[i] = a[i] + c[i];
   b[i+1] = b[i+1]+a[i];
}
a[N-1] = a[N-1]+c[N-1];
```

- A flow dependence that can in general not be remedied is a recurrence:

```
for(i=1;i< N; i++)
{
   z[i] = z[i] + l[i]*z[i-1];
}
```

# Recurrence: LU Factorization of Tridiagonal Matrix

$$
\begin{pmatrix}
a_0 & c_0 & & & & \\
b_1 & a_1 & c_1 & & & \\
 & b_2 & a_2 & c_2 & & \\
 & & b_3 & a_3 & c_3 & \\
 & & & b_4 & a_4 & c_4 \\
 & & & & b_5 & a_5
\end{pmatrix}
= \begin{pmatrix}
1 & & & & & \\
\ell_1 & 1 & & & & \\
 & \ell_2 & 1 & & & \\
 & & \ell_3 & 1 & & \\
 & & & \ell_4 & 1 & \\
 & & & & \ell_5 & 1
\end{pmatrix}
\begin{pmatrix}
d_0 & c_0 & & & & \\
 & d_1 & c_1 & & & \\
 & & d_2 & c_2 & & \\
 & & & d_3 & c_3 & \\
 & & & & d_4 & c_4 \\
 & & & & & d_5
\end{pmatrix}
$$

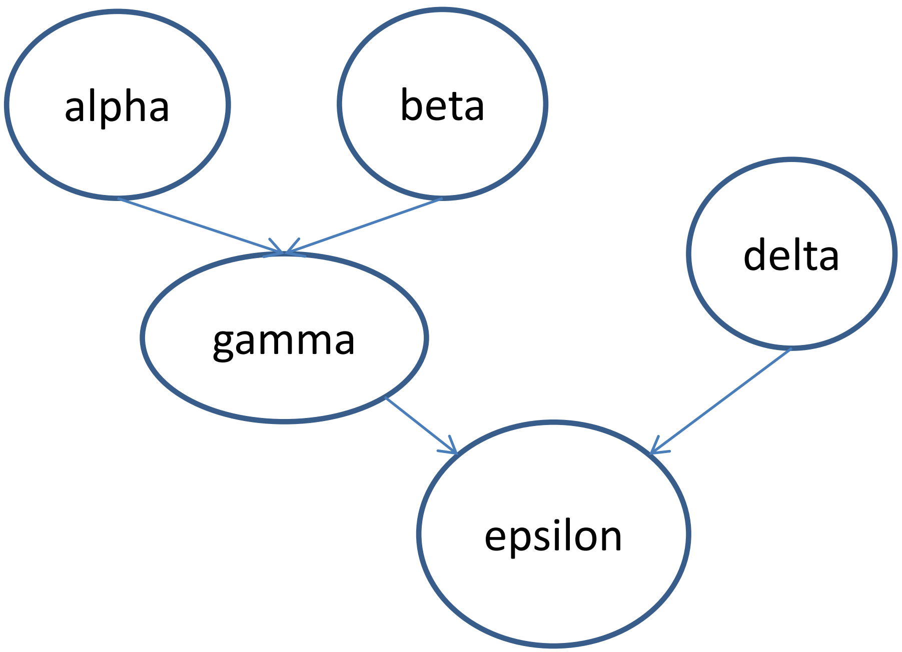
$T = LU$

- T**x**=LU**x**=L**z**=**b**,  **z**=U**x**.
- Proceed as follows:
- L**z**=**b**,  U**x**=**z**
- L**z**=**b** is solved by:

```
z[0] = b[0];
for(i=1;i< n; i++)
{
   z[i] = b[i] - l[i]*z[i-1];
}
```

- Cyclic reduction probably is the best method to solve tridiagonal systems
- Z. Liu, B. Chapman, Y. Wen and L. Huang. *Analyses for the Translation of OpenMP Codes into SPMD Style with Array Privatization*. OpenMP shared memory parallel programming: International Workshop on OpenMP
- C. Addison, Y. Ren and M. van Waveren. *OpenMP Issues Arising in the Development of Parallel BLAS and LAPACK libraries*. J. Sci. Programming – OpenMP, 11(2), 2003.
- S.F. McGinn and R.E. Shaw. Parallel Gaussian Elimination Using OpenMP and MPI

```
V=alpha();
W=beta();
X=gamma(v,w);
Y=delta();
printf("%g\n", epsilon(x,y));
```




Data dependence diagram

Functions alpha, beta, delta may be executed in parallel

# Worksharing sections Directive

sections directive enables specification of task parallelism

- Sections construct gives a different structured block to each thread.

```
#pragma omp sections [clause list]
                     private (list)
                     firstprivate (list)
                     lastprivate (list)
                     reduction (operator: list)
                     nowait
{
#pragma omp section
    structured_block
#pragma omp section
    structured_block
}
```

```c
#include "omp.h"
#define N 1000
int main(){
   int i;
   double a[N], b[N], c[N], d[N];
   for(i=0; i<N; i++){
      a[i] = i*2.0;
      b[i] = i + a[i]*22.5;
   }
   #pragma omp parallel  shared(a,b,c,d) private(i)
   {
      #pragma omp sections nowait
      {
         #pragma omp section
            for(i=0; i<N;i++) c[i] = a[i]+b[i];
         #pragma omp section
            for(i=0; i<N;i++) d[i] = a[i]*b[i];
       }
   }
}
```

Two tasks are computed concurrently

By default, there is a barrier at the end of the sections. Use the "nowait" clause to turn of the barrier.

```
#include “omp.h”

#pragma omp parallel
{
#pragma omp sections
      {
         #pragma omp section
            v=alpha();
         #pragma omp section
            w=beta();
      }
#pragma omp sections
      {
         #pragma omp section
            x=gamma(v,w);
         #pragma omp section
            y=delta();
      }
      printf(“%g\n”, epsilon(x,y));
}
```

# Synchronization I

- Threads communicate through shared variables. Uncoordinated access of these variables can lead to undesired effects.
  - E.g. two threads update (write) a shared variable in the same step of execution, the result is dependent on the way this variable is accessed. This is called a race condition.
- To prevent race condition, the access to shared variables must be synchronized.
- Synchronization can be time consuming.
- The barrier directive is set to synchronize all threads. All threads wait at the barrier until all of them have arrived.

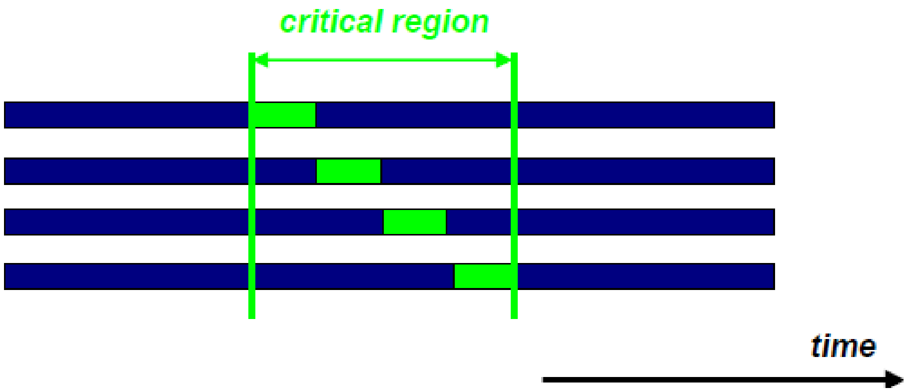
# Synchronization II

- Synchronization imposes order constraints and is used to protect access to shared data
- High level synchronization:
  - critical
  - atomic
  - barrier
  - ordered
- Low level synchronization
  - flush
  - locks (both simple and nested)

# Synchronization: critical

- Mutual exclusion: only one thread at a time can enter a critical region.

```
{
   double res;
   #pragma omp parallel
   {
       double B;
       int  i, id, nthrds;
       id = omp_get_thread_num();
       nthrds = omp_get_num_threads();
       for(i=id; i<niters; i+=nthrds){
           B = some_work(i);
           #pragma omp critical
           consume(B,res);
       }
   }
}
```




Threads wait here: only one thread at a time calls consume(). So this is a piece of sequential code inside the for loop.

```
sum = 0;
#pragma omp parallel shared(n,a,sum) private(TID,sumLocal)
  {
     TID = omp_get_thread_num();
     sumLocal = 0;
     #pragma omp for
       for (i=0; i<n; i++)
          sumLocal += a[i];
     #pragma omp critical (update_sum)
      {
       sum += sumLocal;
       printf("TID=%d: sumLocal=%d sum = %d\n",TID,sumLocal,sum);
      }
  } /*-- End of parallel region --*/
```

```
{
  ...
   #pragma omp parallel
   {
       #pragma omp for nowait shared(best_cost)
       for(i=0; i<N; i++){
          int   my_cost;
          my_cost = estimate(i);
          #pragma omp critical
          {
             if(best_cost < my_cost)
                best_cost = my_cost;
          }
       }
   }
}
```

Only one thread at a time executes if() statement. This ensures mutual exclusion when accessing shared data. Without critical, this will set up a **race condition**, in which the computation exhibits nondeterministic behavior when performed by multiple threads accessing a shared variable

# Synchronization: atomic

- atomic provides mutual exclusion but only applies to the load/update of a memory location.
- This is a lightweight, special form of a critical section.
- It is applied only to the (single) assignment statement that immediately follows it.

```
{
  …
  #pragma omp parallel
  {
        double tmp, B;
        ….
        #pragma omp atomic
        {
           X+=tmp;
        }
  }
}
```

Atomic only protects the update of X.

```
int ic, i, n;
ic = 0;
#pragma omp parallel shared(n,ic) private(i)
   for (i=0; i++, i<n)
    {
      #pragma omp atomic
         ic = ic + 1;
    }
```

"ic" is a counter. The atomic construct ensures that no updates are lost when multiple threads are updating a counter value.

- Atomic construct may only be used together with an expression statement with one of operations: +, *, -, /, &, ^, |, <<, >>.

```
int ic, i, n;
ic = 0;
#pragma omp parallel shared(n,ic) private(i)
   for (i=0; i++, i<n)
    {
      #pragma omp atomic
        ic = ic + bigfunc();
    }
```

- The atomic construct does not prevent multiple threads from executing the function bigfunc() at the same time.

# Synchronization: barrier

Suppose each of the following two loops are run in parallel over i, this may give a wrong answer.

```
for(i= 0; i<N; i++)
   a[i] = b[i] + c[i];
for(i= 0; i<N; i++)
   d[i] = a[i] + b[i];
```

There could be a data race in a[].

```
for(i= 0; i<N; i++)
    a[i] = b[i] + c[i];
for(i= 0; i<N; i++)
    d[i] = a[i] + b[i];
```

wait

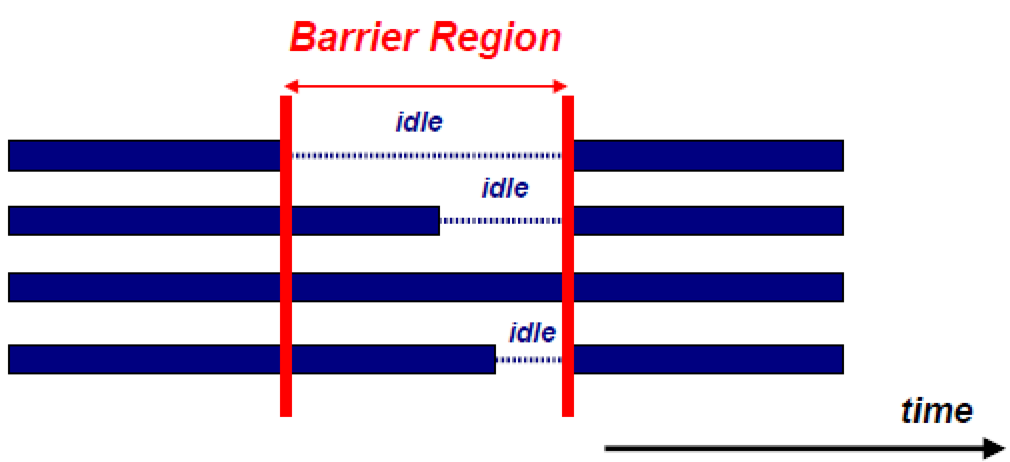
barrier

To avoid race condition:

- NEED: All threads wait at the barrier point and only continue when all threads have reached the barrier point.

Barrier syntax:

- #pragma omp barrier



# Synchronization: barrier

barrier: each threads waits until all threads arrive

```
#pragma omp parallel shared (A,B,C) private (id)
{
    id=omp_get_thread_num();
    A[id] = big_calc1(id);
    #pragma omp barrier
    #pragma omp for
        for(i=0; i<N;i++){C[i]=big_calc3(i,A);}
    #pragma omp for nowait
        for(i=0;i<N;i++) {B[i]=big_calc2(i,C);}
    A[id]=big_calc4(id);
}
```

Implicit barrier at the end of for construct

No implicit barrier due to nowait

Implicit barrier at the end of a parallel region

# When to Use Barriers

- If data is updated asynchronously and data integrity is at risk
- Examples:
  - Between parts in the code that read and write the same section of memory
  - After one timestep/iteration in a numerical solver
- Barriers are expensive and also may not scale to a large number of processors

# “master” Construct

- The “master” construct defines a structured block that is only executed by the master thread.
- The other threads skip the “master” construct. No synchronization is implied.
- It does not have an implied barrier on entry or exit.
- The lack of a barrier may lead to problems.

```
#pragma omp parallel
{
    …
    #pragma omp master
    {
        exchange_information();
    }
    #pragma omp barrier
    …
}
```

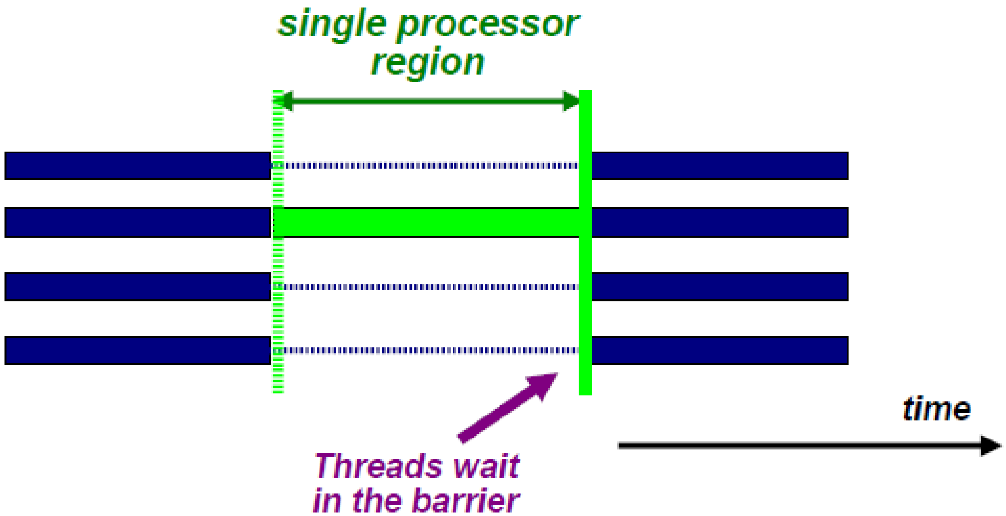
```
#pragma omp parallel shared(a,b) private(i)
 {
    #pragma omp master
     {
       a = 10;
       printf("Master construct is executed by thread %d\n",
               omp_get_thread_num());
     }

    #pragma omp barrier

    #pragma omp for
     for (i=0; i<n; i++)
         b[i] = a;

} /*-- End of parallel region --*/

printf("After the parallel region:\n");
for (i=0; i<n; i++)
     printf("b[%d] = %d\n",i,b[i]);
```

- Master construct to initialize the data

# “single” Construct

- The “single” construct builds a block of code that is executed by only one thread (not necessarily the master thread).
- A barrier is implicitly set at the end of the single block (the barrier can be removed by the *nowait* clause)

```
#pragma omp parallel
{
    …
    #pragma omp single
    {
        exchange_information();
    }
    do_other_things();
    …
}
```



```
#pragma omp parallel shared(a,b) private(i)
{
   #pragma omp single
   {
      a = 10;
      printf("Single construct executed by thread %d\n",
             omp_get_thread_num());
   }
   /* A barrier is automatically inserted here */

   #pragma omp for
   for (i=0; i<n; i++)
       b[i] = a;

} /*-- End of parallel region --*/

   printf("After the parallel region:\n");
   for (i=0; i<n; i++)
       printf("b[%d] = %d\n",i,b[i]);
```

- Single construct to initialize a shared variable

# Synchronization: ordered

- The “ordered” region executes in the sequential order

```
#pragma omp parallel private (tmp)
{
    …
    #pragma omp for ordered reduction(+:res)
    for(i=0;i<N;i++)
    {
        tmp = compute(i);
     #pragma ordered
        res += consum(tmp);
    }
    do_other_things();
    …
}
```

# Synchronization: Lock routines

- A lock implies a memory fence of all thread visible variables.
- These routines are used to guarantee that only one thread accesses a variable at a time to avoid race conditions.
- C/C++ lock variables must have type “omp_lock_t” or “omp_nest_lock_t”.
- All lock functions require an argument that has a pointer to omp_lock_t or omp_nest_lock_t.
- Simple Lock routines:
  - omp_init_lock(omp_lock_t*); omp_set_lock(omp_lock_t*);

  omp_unset_lock(omp_lock_t*);

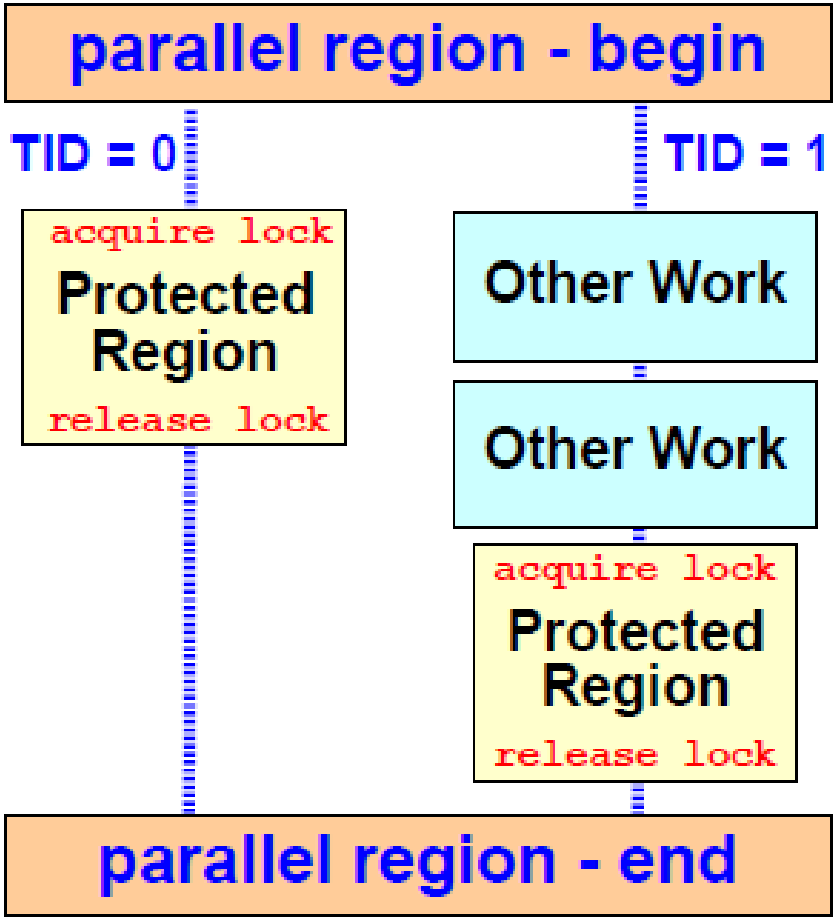
  omp_test_lock(omp_lock_t*); omp_destroy_lock(omp_lock_t*);

  [http://gcc.gnu.org/onlinedocs/libgomp/index.html#Top](http://gcc.gnu.org/onlinedocs/libgomp/index.html#Top)

# General Procedure to Use Locks

1. Define the lock variables
2. Initialize the lock via a call to omp_init_lock
3. Set the lock using omp_set_lock or omp_test_lock. The latter checks whether the lock is actually available before attempting to set it. It is useful to achieve asynchronous thread execution.
4. Unset a lock after the work is done via a call to omp_unset_lock.
5. Remove the lock association via a call to omp_destroy_lock.

# Locking Example




- The protected region contains the update of a shared variable
- One thread acquires the lock and performs the update
- Meanwhile, other threads perform some other work
- When the lock is released again, the other threads perform the update

```
omp_lock_t lck;
omp_init_lock(&lck);
#pragma omp parallel shared(lck) private (tmp, id)
{
    id = omp_get_thread_num();
    tmp = do_some_work(id);
    omp_set_lock(&lck);
       printf(“%d %d\n”, id, tmp);
    omp_unset_lock(&lck);
}
omp_destroy_lock(&lck);
```

Initialize a lock associated with lock variables “lck” for use in subsequent calls.

Thread waits here for its turn.

Release the lock so that the next thread gets a turn

Dissociate the given lock variable from any locks.

# Runtime Library Routines

- Routines for modifying/checking number of threads
  - omp_set_num_threads(int n);
  - int omp_get_num_threads(void);
  - int omp_get_thread_num(void);
  - int omp_get_max_threads(void);
- Test whether in active parallel region
  - int omp_in_parallel(void);
- Allow system to dynamically vary the number of threads from one parallel construct to another
  - omp_set_dynamic(int set)
    - set = true: enables dynamic adjustment of team sizes
    - set = false: disable dynamic adjustment
  - int omp_get_dynamic(void)
- Get number of processors in the system
  - int omp_num_procs(void); returns the number of processors online

http://gcc.gnu.org/onlinedocs/libgomp/index.html#Top

# Default Data Storage Attributes

- A shared variable has a single storage location in memory for the whole duration of the parallel construct. All threads that reference such a variable accesses the same memory. Thus, reading/writing a shared variable provides an easy mechanism for communicating between threads.
  - In C/C++, by default, all program variables except the loop index become shared variables in a parallel region.
  - Global variables are shared among threads
  - C: File scope variables, static variables, dynamically allocated memory (by malloc(), or by new).
- A private variable has multiple storage locations, one within the execution context of each thread.
  - Not shared variables
    - Stack variables in functions called from parallel regions are private.
    - Automatic variables within a statement block are private.
  - This holds for pointer as well. Therefore, do not assign a private pointer the address of a private variable of another thread. The result is not defined.

```c
/** main file **/
#include <stdio.h>
#include <stdlib.h>

double  A[100];
int main(){
    int index[50];
    #pragma omp parallel
       work(index);
    printf("%d\n", index[0]);
}
```

```c
/** file 1 **/
#include <stdio.h>
#include <stdlib.h>

extern double  A[100];
void work(int *index){
    double temp[50];
    static int count;
}
```

- Variables “A”, “index” and “count” are shared by all threads.
- Variable “temp” is local (or private) to each thread.

# Changing Data Storage Attributes

- Clauses for changing storage attributes
  - “shared”, “private”, “firstprivate”
- The final value of a private inside a parallel “for” loop can be transmitted to the shared variable outside the loop with:
  - “lastprivate”
- The default attributes can be overridden with:
  - Default(private|shared|none)
- All data clauses listed here apply to the parallel construct region and worksharing construct region except “shared”, which only applies to parallel constructs.

# Private Clause

- "private (variable list)" clause creates a new local copy of variables for each thread.
  - Values of these variables are not initialized on entry of the parallel region.
  - Values of the data specified in the private clause can no longer be accessed after the corresponding region terminates (values are not defined on exit of the parallel region).

```
/*** wrong implementation ***/
int main(){
    int  tmp = 0;
#pragma omp parallel for private(tmp)
    for (int j=0; j<1000;j++)
        tmp += j;
    printf("%d\n", tmp);
}
```

"tmp" is not initialized

"tmp" is 0  in version 3.0; unspecified in version 2.5.

# Firstprivate Clause

- firstprivate initializes each private copy with the corresponding value from the master thread.

```
/*** still wrong implementation ***/
int main(){
    int  tmp = 0;
#pragma omp parallel for firstprivate(tmp)
    for (int j=0; j<1000;j++)
        tmp += j;
    printf("%d\n", tmp);
}
```

Each thread get its own "tmp" with an initial value of 0.

"tmp" is 0 in version 3.0; unspecified in version 2.5.

# Lastprivate Clause

- Lastprivate clause passes the value of a private variable from the last iteration to a global variable.
  - It is supported on the work-sharing loop and sections constructs.
  - It ensures that the last value of a data object listed is accessible after the corresponding construct has completed execution.
  - In case use with a work-shared loop, the object has the value from the iteration of the loop that would be last in a "sequential" execution.

```
/*** useless implementation ***/
int main(){
    int  tmp = 0;
#pragma omp parallel for firstprivate(tmp) lastprivate(tmp)
    for (int j=0; j<5;j++)
        tmp += j;
    printf("%d\n", tmp);
}
```

"tmp" is defined as its value at the "last sequential" iteration, i.e, *j* = 5.

# Correct Usage of Lastprivate

```
/*** correct usage of lastprivate ***/
int main(){
    int  a, j;
#pragma omp parallel for private(j) lastprivate(a)
    for (j=0; j<5;j++)
    {
        a = j + 2;
        printf("Thread %d has a value of a = %d for j =  %d\n",
                omp_get_thread_num(), a, j);
     }
     printf("value of a after parallel = %d\n", a);
}
```

```
Tread 0 has a value of a = 2 for j =  0
Tread 2 has a value of a = 4 for j =  2
Tread 1 has a value of a = 3 for j =  1
Tread 3 has a value of a = 5 for j =  3
Tread 4 has a value of a = 6 for j =  4
value of a after parallel = 6
```

# Default Clause

- C/C++ only has default(shared) or default(none)
- Only Fortran supports default(private)
- Default data attribute is default(shared)
  - Exception: #pragma omp task
- Default(none): no default attribute for variables in static extent. Must list storage attribute for each variable in static extent. Good programming practice.

# Lexical (static) and Dynamic Extent I

- Parallel regions enclose an arbitrary block of code, sometimes including calls to another function.
- The lexical or static extent of a parallel region is the block of code to which the parallel directive applies.
- The dynamic extent of a parallel region extends the lexical extent by the code of functions that are called (directly or indirectly) from within the parallel region.
- The dynamic extent is determined only at runtime.

# Lexical and Dynamic Extent II

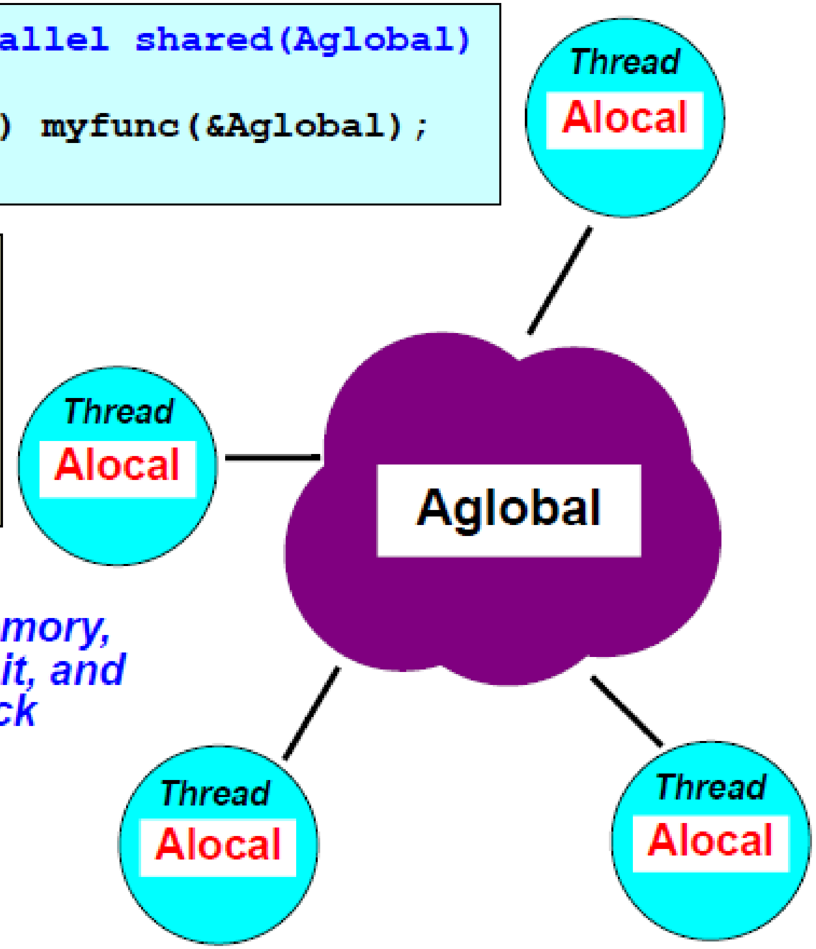
```
int main(){
#pragma omp parallel
    {
        print_thread_id();
     }
}

void print_thread_id()
{
    int id = omp_get_thread_num();
    printf("Hello world from thread %d\n", id);
}
```

Static extent

Dynamic extent

```
#omp parallel shared(Aglobal)
{
    (void) myfunc(&Aglobal);
}
```

```
void myfunc(float *Aglobal)
{
    int Alocal;

    .......
}
```




*Variable **Alocal** is in private memory, managed by the thread owning it, and stored on the so-called stack*

```
void caller(int *a, int n) {
int i,j,m=3;
#pragma omp parallel for
for (i=0; i<n; i++) {
  int k=m;
  for (j=1; j<=5; j++) {
    callee(&a[i], &k, j);
  }
}
void callee(int *x, int *y, int
   z) {
  int ii;
  static int cnt;
  cnt++;
  for (ii=1; ii<z; ii++) {
    *x = *y + z;
  }
}
```

| Var | Scope | Comment |
|---|---|---|
| a | shared | Declared outside parallel construct |
| n | shared | same |
| i | private | Parallel loop index |
| j | shared | Sequential loop index |
| m | shared | Declared outside parallel construct |
| k | private | Automatic variable/parallel region |
| x | private | Passed by value |
| *x | shared | (actually a) |
| y | private | Passed by value |
| *y | private | (actually k) |
| z | private | (actually j) |
| ii | private | Local stack variable in called function |
| cnt | shared | Declared static (like global) |

R. Hartman-Baker. Using OpenMP

# Threadprivate

- Threadprivate makes global data private to a thread
  - C/C++: file scope and static variables, static class members
  - Each thread gives its own set of global variables, with initial values undefined.
- Different from private
  - With private clause, global variables are masked.
  - Threadrpivate preserves global scope within each thread.
  - Parallel regions must be executed by the same number of threads for global data to persist.
- Threadprivate variables can be initialized using copyin clause or at time of definition.

If all of the conditions below hold, and if a threadprivate object is referenced in two consecutive (at run time) parallel regions, then threads with the same thread number in their respective regions reference the same copy of that variable:

- Neither parallel region is nested inside another parallel region.
- The number of threads used to execute both parallel regions is the same.

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

int   *pglobal;
#pragma omp threadprivate(pglobal)

int main(){
    ...
#pragma omp parallel for private(i,j,sum,TID) shared(n,length,check)
    for (i=0; i<n;i++)
    {
        TID = omp_get_thread_num();
        if((pglobal = (int*) malloc(length[i]*sizeof(int))) != NULL) {
            for(j=sum=0; j < length[i];j++) pglobal[j] = j+1;
            sum = calculate_sum(length[i]);
            printf("TID %d: value of sum for I = %d is %d\n", TID,i,sum);
            free(pglobal);
        } else {
            printf("TID %d: not enough memory : length[%d] = %d\n", TID,i,length[i]);
        }
    }
}
```

Threadprivate directive is used to give each thread a private copy of the global pointer pglobal.

```
/* source of function calculate_sum() */
extern int *pglobal;

int  calculate_sum(int   length){
    int  sum = 0;
    for (j=0; j<length;j++)
    {
        sum  += pglobal[j];
     }
     return (sum);
}
```

```
#include <omp.h>
static int sum0=0;
#pragma omp threadprivate (sum0)
int main()
{ int sum = 0;
  int i ;
  . . .
  for ( . . . )
#pragma omp parallel
   {
  sum0 = 0;
  #pragma omp for
    for ( i = 0; i <= 1000; i++)
       sum0 = sum0 + . . .
  #pragma omp critical
      sum = sum + sum0 ;
} /* end of parallel region */
```

- Each thread has its own copy of sum0, updated in a parallel region that is called several times. The values for sum0 from one execution of the parallel region will be available when it is next started.

# Copyin Clause

- Copyin allows to copy the master thread's threadprivate variables to corresponding threadprivate variables of the other threads.

```
int   global[100];
#pragma omp threadprivate(global)

int main(){
   for(int i= 0; i<100; i++) global[i] = i+2; // initialize data
#pragma omp parallel copyin(global)
   {
       /// parallel region, each thread gets a copy of global, with initialized value
   }
}
```

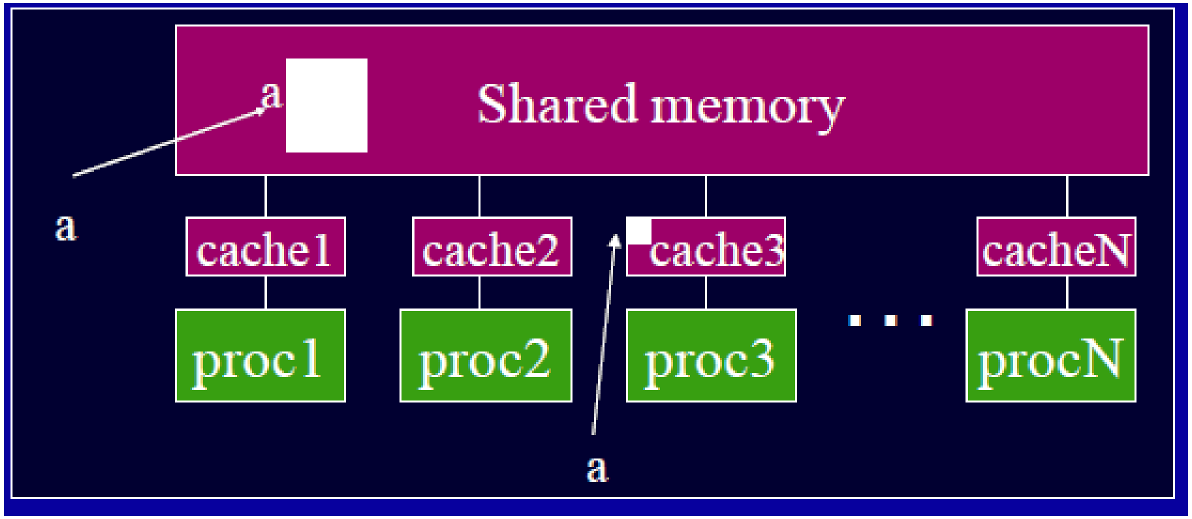
# Copyprivate Clause

- Copyprivate clause is supported on the single directive to broadcast values of privates from one thread of a team to the other threads in the team.
  - The typical usage is to have one thread read or initialize private data that is subsequently used by the other threads as well.
  - After the single construct has ended, but before the threads have left the associated barrier, the values of variables specified in the associated list are copied to the other threads.
  - Do not use copyprivate in combination with the nowait clause.

```
#include “omp.h”
Void input_parameters(int, int); // fetch values of input parameters

int main(){
   int Nsize, choice;
#pragma omp parallel private(Nsize, choice)
   {
       #pragma omp single copyprivate (Nsize, choice)
           input_parameters(Nsize,choice);
      do_work(Nsize, choice);
   }
}
```

# Flush Directive

- OpenMP supports a shared memory model.
  - However, processors can have their own “local” high speed memory, the registers and cache.
  - If a thread updates shared data, the new value will first be saved in register and then stored back to the local cache.
  - The update are thus not necessarily immediately visible to other threads.



# Flush Directive

The flush directive is to make a thread's temporary view of shared data consistent with the value in memory.

- #pragma omp flush (list)
- Thread-visible variables are written back to memory at this point.
- For pointers in the list, note that the pointer itself is flushed, not the object it points to.

References:


- http://bisqwit.iki.fi/story/howto/openmp/
- http://openmp.org/mp-documents/omp-hands-on-SC08.pdf
- https://computing.llnl.gov/tutorials/openMP/
- http://www.mosaic.ethz.ch/education/Lectures/hpc
- R. van der Pas. An Overview of OpenMP
- B. Chapman, G. Jost and R. van der Pas. Using OpenMP: Portable Shared Memory Parallel Programming. The MIT Press, Cambridge, Massachusetts, London, England
- B. Estrade, Hybrid Programming with MPI and OpenMP